VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

BEZSNÍMKOVÉ RENDEROVÁNÍ

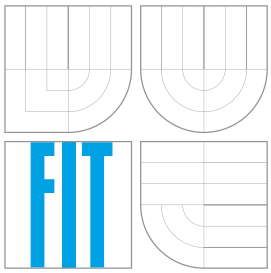
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VOJTĚCH KRUPIČKA

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

BEZSNÍMKOVÉ RENDEROVÁNÍ

FRAMELESS RENDERING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VOJTĚCH KRUPIČKA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ HAVEL

BRNO 2012

Abstrakt

Tato diplomová práce se zabývá problémem zobrazovaní počítačové grafiky v reálném čase s využitím metody *bezsnímkového renderování* jako protipólu k tradičnímu způsobu, který je založen na přepínání výstupu mezi dvěma buffery. Metoda *bezsnímkového renderování* je zkoumána a definována do větší hloubky a detailně popsána její *adaptivní* varianta, která přináší kvalitnější výstup bez výraznějšího snížení odezvy. Dále tato práce popisuje implementaci aplikace, která byla vyvíjena pro demonstraci principu a funkčnosti metody *bezsnímkového renderování* na vybraných scénách a vyhodnocení prováděných testů se zaměřením na kvalitu výstupu.

Abstract

This master's thesis deals with the problem of real-time rendering of computer graphics using the method of *frameless rendering* as counterpart to the traditional way, which is based on switching between two output buffers. *Frameless rendering* method is defined and studied in greater depth and its *adaptive* variant, which delivers better output quality without a significant reduction of latency, is described in detail. In addition, this thesis describes the implementation of the application which has been developed to demonstrate the principle and functionality of the *frameless rendering* on the selected scenes. It also includes evaluation of performed tests focused to the output quality.

Klíčová slova

Počítačová grafika, 3D grafika, zobrazování v reálném čase, bezsnímkové renderování, raytracing, OOP, C++, demostrační aplikace pro bezsnímkové renderování

Keywords

Computer graphics, 3D graphic, real-time rendering of computer graphics, frameless rendering, raytracing, OOP, C++, demo aplication for frameless rendering

Citace

Vojtěch Krupička: Bezsnímkové renderování, diplomová práce, Brno, FIT VUT v Brně, 2012

Bezsnímkové renderování

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně pod vedením Ing. Jiřího Havla. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vojtěch Krupička
18. května 2012

Poděkování

Děkuji Ing. Jiřímu Havlovi za cenné rady a připomínky při návrhu demonstrační aplikace a při psaní této tiskové zprávy. Dále děkuji Thomasi Schlömerovi za radu při implementaci rekonstrukce a také děkuji všem, kteří mě při práci podporovali a pomáhali.

Obsah

Kapitola 1

Úvod

V počítačové grafice se po dlouhou řadu let zabydlel koncept zobrazování grafického výstupu na zobrazovací zařízení (monitor, displej, apod.), který využívá existenci dvou *bufferů*. V jednom z nich je uložen aktuální grafický výstup, jenž je současně zobrazen na displeji, zatímco do druhého probíhá výpočet nového snímku. Než bude nový snímek kompletně celý spočítán a zapsán do druhého, skrytého *bufferu*, nebude na výstupním zařízení docházet k žádným změnám. Jakmile je výpočet dokončen, dojde k výměně obou *bufferů* a nový snímek je zobrazen na displeji. Tato činnost se děje opakovaně. Její zásadní nevýhodou ale je, že výpočet snímku trvá určitý čas t a aktuální snímek zobrazený na displeji velmi rychle stárne. A pokud jsou uživatelské vstupy příliš rychlé — rychlejší než čas t, je jasné, že grafický výstup bude nepříjemně zpomalený a nebude odpovídat skutečnosti.

Způsobů, jak řešit problém s odezvou, resp. s dobou výpočtu nového snímku, je mnoho. Většina z nich zpravidla vede ke snižování komplexnosti zobrazované scény (např. snižování počtu primitiv ve scéně) nebo ke snižování rozlišení výsledného výstupu. To ale má přímý dopad na kvalitu výstupu samotného. Je-li i přesto čas t příliš veliký, ve výstupní animaci nebo video-sekvenci bude s největší pravděpodobností docházet k trhání. Není těžké si představit jak snímky v tomto klasickém pojetí s využitím dvou bufferů stárnou. V okamžiku, kdy je jeden snímek dokončen a zobrazen na displeji je již t jednotek starý. Na displeji ovšem zůstává dalších t jednotek staticky nezměněn, dokud není vypočítán nový snímek. Z toho plyne, že stáří snímků je teoreticky $2t$[1]. V aplikacích, kde je nutná co nejrychlejší odezva na uživatelské vstupy (např. aplikace založené na virtuální realitě, apod.), je potřeba tento problém řešit, nebo se zaměřit na jiné principy zobrazování grafického výstupu. Jedním z těchto principů je *bezsnímkové renderování*.

Bezsnímkové renderování představuje grafické paradigma zcela měnící pohled na tradiční způsob, kterým je zobrazování grafického výstupu téměř výhradně realizováno (viz výše). Tento nový přístup byl poprvé představen již v roce 1994 v [2], kde jeho autoři postulovali, že používání dvou bufferů je koncepčně špatné a navrhovali zaměřit se na aktualizaci výstupu co nejdříve po obdržení nových vstupů, byť by se mělo jednat pouze o malou část snímku. Přestože je tento přístup velmi inovativní, je zarážející, že se jeho rozšíření, definici a studia za 17 let od jeho prvního představení ujalo pouze velmi málo výzkumníků, v největší míře právě lidí okolo původního týmu (viz kapitola 2.2).

Tato práce se snaží přiblížit a vysvětlit metodu *bezsnímkového renderování* a zasadit ji do širšího kontextu v porovnání s klasickým konceptem využívajícím dva *buffery*. Definuje základní pojmy, které jsou nezbytné pro pochopení problematiky a zkoumá všechny

[1] Prakticky tomu tak být nemusí, neboť t může být pro různé snímky různé.

aspekty bezsnímkového paradigmatu, jenž je nutné brát na vědomí. Dále se snaží popsat a postihnout další metody vycházející z původního pojetí *bezsnímkového renderování*, jako je jeho *adaptivní varianta*. Kapitola 2 se zaměřuje na popis vlastního *bezsnímkového renderování*, tak jak bylo představeno v [2] a posléze definováno v [20], kapitola 3 se poté do větší hloubky věnuje jeho *adaptivní* variantě popsané v [6]. Kapitola 4 se zabývá implementací demonstrační aplikace, a poslední kapitola 5 provádí testy a diskutuje dosažené výsledky se zaměřením na jejich kvalitu. Na závěr jsou diskutovány aspekty v této práci opomenuté a možný budoucí vývoj jak aplikace tak i samotné metody.

Kapitola 2

Bezsnímkové renderování

2.1 Motivace

Jak již bylo řečeno v úvodu, *bezsnímkové renderování* (z angl. *frameless rendering*) je nové paradigma v počítačové grafice, které se snaží změnit pohled na klasický přístup[1] pro zobrazování počítačové grafiky. Hlavním cílem *bezsnímkového renderování* (viz [2]) je snaha snížit odezvu na uživatelské vstupy při zachování kvalitního grafického výstupu. Toho je možné dosáhnout výběrem vhodné sady pixelů, jejich aktualizací a okamžitým zobrazením na výstup. A to i přesto, že v daném okamžiku nemusí být všechny hodnoty „nových“ pixelů spočítány. Důsledkem toho je naprosto odlišné vnímání tvorby počítačové grafiky, jako zobrazování snímků na displeji v reálném čase.

Prvním předpokladem, jak je uvedeno v [2], je skutečnost, že nedochází k vykreslování obrazu zleva doprava po jednotlivých pixelech a shora dolů po jednotlivých řádcích. A za druhé, jejich přístup umožňuje získat aktuální grafický výstup v jakémkoliv časovém okamžiku. Tedy vždy, když je to nezbytně nutné (na základě změny stavu aplikace nebo jako reakci na uživatelské vstupy). Z toho vyplývá, že neexistuje žádná sekvence snímků tvořících animaci, žádný počet zobrazených snímků za vteřinu a dokonce ani žádný konkrétní celistvý snímek. Proto název *bezsnímkové renderování*.

V [2] vycházejí z jednoduché premisy, která říká, že pro standardní obnovovací frekvenci současných displejů ($60Hz$ a více), není nezbytně nutné pokaždé přepočítávat a překreslovat celý výstup, jež má být zobrazen. Nejen že to není nutné, ale ani lidský mozek na to není schopen reagovat[2]. Naopak stačí, aby bylo aktualizováno dostatečné množství pixelů, které aktualizaci vyžadují, a pokud bude tato sada dostatečně malá a výpočet dostatečně rychlý, lze snížit odezvu a zcela opustit koncept využívající dvou *bufferů*. Odvětví počítačové grafiky, do kterého *bezsnímkové renderování* lze spíše zařadit, a kde může využít svůj potenciál naplno, tedy neleží mezi klasickými displeji (ty na tento koncept nejsou ani technicky stavěny), ale spíše v oblasti systémů založených na vnímání a virtuální realitě. Cílem není přesná simulace chování kamery jako elektronického zařízení, ale naopak snaha popsat a simulovat princip činnosti lidského oka. Jak je ukázáno v [23], lidské vnímání okolního světa zrakem je nejintenzivnější uprostřed zorného pole, kdežto k jeho okrajům slábne, čehož lze využít např. u velkých displejů, nebo při použití *head mounted* displejů.

[1] V dalším textu se bude *klasickým přístupem* rozumět přístup využívající právě dva *buffery*.

[2] Při animacích vzniká tzv. *motion smear*, podle kterého je lidský mozek schopen rozlišovat impulsy přicházející na sítnici oka minimálně každých $125ms$ (viz [3]).

(a) Ideální obraz (b) Tradiční *bezsnímková* metoda (c) *Adaptivní* varianta

Obrázek 2.1: Obrázek srovnávající ideální výstup získaný *raytracerem* s výstupem základního *besnímkového rendereru* a s výstupem *adaptivní* varianty, která využívá řízeného vzorkování a rekonstrukce obrazu. Zatímco výstup základní metody obsahuje množství artefaktů, její adaptivní varianta poskytuje relativně věrný výstup bez artefaktů a s automatickým vyhlazením hran. A to při stejném počtu aktualizovaných pixelů v jednom kroku (kterých v tomto případě bylo 9 000, tedy méně než 14% všech pixelů v rastru).

2.2 Historie a předchozí výzkum

Paradigma *bezsnímkového renderování* vzniká v roce 1994 a vychází z [2], ve kterém byla nastíněna úvaha zamýšlející se nad tím, proč je princip používání dvou *bufferů* špatný. Žádné výraznější definování a formalizování metody provedeno nebylo. Od té doby vyšlo pouze několik publikací, které se *bezsnímkovému renderování* věnují ve větší míře, nejvíce pak [20], kde jsou položeny formální základy této metody a definovány a popsány různé aspekty (např. faktory podílející se na vzniku artefaktů). Další publikace autorky *Ellen J. Scher Zieger* následují vzápětí – [21], [22] a [23]. A to bylo na dlouhou dobu vše. Na tyto práce bylo navázáno o pět let později v [4] a [5], což byla příprava pro tzv. *adaptivní variantu* publikovanou v [6]. Její autoři se snaží rozšířit klasické pojetí *bezsnímkového renderování* o tzv. *řízené vzorkování*, které vybírá pixely pro aktualizaci v místech nejpravděpodobnější změny (např. hrany objektů), a následnou *rekonstrukci obrazu* (více viz kapitola 3). Poslední dostupnou literaturou jsou dvě diplomové práce [17] a [14] na toto téma.

Tato práce vychází právě z článků a děl zmíněných v předchozím odstavci a pokouší se vše spojit do jednotného uceleného textu, který by měl čtenáři poskytnou úplné informace o metodě *bezsnímkového renderování*. Kromě toho byla značná část úsilí věnována návrhu a implementaci vlastní aplikace pro demonstraci tohoto paradigmatu.

2.3 Srovnání s klasickým přístupem

Jak bylo nastíněno v úvodní kapitole, klasický přístup trpí především neaktuálností zobrazovaného výstupu na displeji. Výpočet snímku, který je zapisován do pomocného *bufferu*, trvá určitý čas t. Aby ale mohl vlastní výpočet začít, je třeba „uzamknout“ aktuální stav aplikace a ten se po dobu t nemění. V okamžiku, kdy je celý snímek spočítán, je již t jednotek starý. V tomto okamžiku je zobrazen na displeji (výměnou obou *bufferů* v paměti grafické karty), kde setrvá dalších t jednotek, než je vypočítán následující snímek. Celkové

stáří každého snímku je tedy $2t$. Princip činnosti klasického přístupu je popsán pomocí algoritmu 2.1.

Algoritmus 2.1: Pseudokód pro klasický přístup s využitím dvou *bufferů*.

```
1 alokuj paměť pro frameBuffer o rozměrech X × Y;
2 forever do
3 |   for x ← 0 to X do
4 |   |   for y ← 0 to Y do
5 |   |   |   frameBuffer(x, y) ← spočítejBarvuNa(x, y);
6 |   |   end
7 |   end
8 |   prohoď oba buffery na grafické kartě a zobraz aktuální;
9 end
```

Jedná se o velmi jednoduchý algoritmus, který nejprve alokuje dostatečné množství paměti pro uložení rastru do dvojrozměrného pole `frameBuffer` o daných rozměrech X a Y. Poté v nekonečné smyčce (na řádku 2), resp. dokud přicházejí uživatelské vstupy nebo není aplikace explicitně ukončena, počítá hodnoty pixelů pro každý řádek zleva doprava a odshora dolů. Jakmile jsou hodnoty všech pixelů spočítané a zapsané do `frameBufferu`, je možné jeho obsah zobrazit na výstup (výměnou obou *bufferů* na grafické kartě). Pokud ale vnořené smyčky z řádků 3 a 4 trvají příliš dlouho, je odezva pomalá a čím pomalejší odezva bude, tím méně celých snímků bude zobrazeno za jednotku času. To může způsobit trhání animace, které může být postřehnutelné lidským okem.

Princip *bezsnímkového renderování* je však zcela jiný a je nastíněn v algoritmu 2.2. Obě vnořené smyčky vypadly, stejně tak nutnost alokace *bufferu*. V každé iteraci hlavní, nekonečné smyčky jsou nejprve na řádku 2 náhodně vybrány dvě souřadnice x a y, pro které je na řádku 3 spočítána barva. Ta je poté okamžitě zobrazena na výstupu. Odezva tedy závisí pouze na složitosti spočítání barvy jednoho pixelu funkcí `spočítejBarvuNa()`. Algoritmus 2.2 představuje ideální řešení *bezsnímkového renderování*. Bohužel dnešní grafické karty ani zobrazovací zařízení nejsou na takovýto přístup stavěny a neumějí změny promítat na výstup okamžitě. Z toho plyne omezení, se kterým je nutné při implementaci počítat. Algoritmus 2.2 lze upravit tak, aby vyhovoval dnešním technologiím. Jeho podoba je zobrazena v algoritmu 2.3.

Algoritmus 2.2: Pseudokód pro *bezsnímkové renderování*, který vychází z jeho hlavní myšlenky.

```
1 forever do
2 |   (x, y) ← získejNáhodnýPixel();
3 |   barva ← spočítejBarvuNa(x, y);
4 |   pošli novou barvu na displej na souřadnice (x, y);
5 end
```

Algoritmus 2.3 již zohledňuje dnešní technologie. Opět se vrátil `frameBuffer`, jehož přítomnost je bohužel nezbytná. Vnitřní `while` cyklus počítá na řádcích 4 a 5 novou barvu náhodně vybraného pixelu, stejně jako v algoritmu 2.2, dokud není nutné tento cyklus zastavit a tím vynutit překreslení obrazovky (opět záměnou obou *bufferů*). Terminaci `while`

Algoritmus 2.3: Pseudokód pro *bezsnímkové renderování*, který bere v úvahu dnešní technologie grafických karet a zobrazovacích zařízení.

```
1 alokuj paměť pro frameBuffer o rozměrech X × Y;
2 forever do
3 |   while není nutné překreslit displej do
4 |   |   (x, y) ← získejNáhodnýPixel();
5 |   |   frameBuffer(x, y) ← spočítejBarvuNa(x, y);
6 |   end
7 |   prohoď oba buffery na grafické kartě a zobraz aktuální;
8 end
```

cyklu je možné nastavit explicitně (např. každých 0,04s a tím zajistit konstantní zobrazovací frekvenci 25Hz) nebo implicitně (např. při konkrétních uživatelských vstupech, kdy je překreslení nutné – změna pozice kamery, apod.).

Z výše představených algoritmů jsou klíčové obě uvedené funkce `spočítejBarvuNa()` a `získejNáhodnýPixel()`. První z nich má skutečný dopad na celkovou odezvu a počet pixelů, které bude možné aktualizovat v daném časovém okamžiku. Její implementace zpravidla bývá založena na jednoduchém *raytraceru*, který posílá paprsky do scény a počítá výslednou barvu pixelu. Druhá funkce – `získejNáhodnýPixel()` – je již poněkud zajímavější, protože výběr náhodného pixelu sebou přináší faktory, které mají hlavní podíl na výsledné kvalitě výstupu. Více o možných implementacích této funkce viz kapitola 2.7 a kapitola 3.2.

2.4 Definice *bezsnímkového renderování*

Základní princip metody *bezsnímkového renderování* byl již nastíněn v kapitole 2.1 a 2.3. Jedním z nejvýznamnějších kladů této metody je generování grafického výstupu, který je vždy aktuální, byť se zpravidla jedná pouze o jeden pixel, nebo skupinu pixelů. Na druhou stranu je výstup metody znehodnocen výskytem artefaktů v obraze. Metoda *bezsnímkového renderování* totiž provádí časový *supersampling*[3] a prostorový *subsampling*[4].

Časový *supersampling* proto, že pixely jsou zasílány na výstup v menších časových intervalech (častěji než při použití tradiční metody) a prostorový *subsampling* proto, že počet pixelů poslaných na výstup v daném okamžiku zdaleka nepokrývá celé rozlišení displeje. Artefakty v obraze se pak objevují zejména tam, kde dochází k výrazným změnám v geometrii, osvětlení nebo pohybu kamery. Nově aktualizované pixely sice odpovídají skutečnosti, ale jelikož jich je vždy jen nepatrná část, jeví se jako šum rozprostřený přes ostatní, neaktualizované a neplatné pixely. Jedná se o tzv. *asynchronní pixely*, rozptýlené body v rastru, které výrazně zhoršují kvalitu výstupu. Prostorový rozptyl pixelů je označován jako *spatial scatter*.

[3] Technika využívaná v počítačové grafice např. pro odstranění aliasů, která pro jeden pixel použije několik vzorků – subpixelů – a výsledná hodnota – barva – pixelu je průměrnou hodnotou všech vzorků.

[4] Nebo-li *podvzorkování*, tedy pro několik pixelů se použije pouze jeden vzorek.

2.4.1 Princip postupného zdokonalování

Asynchronními pixely jsou označovány všechny platné aktualizované pixely v rastru. Prostorový rozptyl pixelů pak bude větší pokud bude docházet k rapidním změnám geometrie uvnitř scény nebo pokud bude v jednom kroku aktualizováno příliš málo pixelů. Pokud naopak bude scéna statická, nebo se animace zpomalí, výstup by se měl velmi rychle blížit k ideálnímu obrazu. Tento princip je v [20] označen jako *princip postupného zdokonalování*, který spočívá v opakování určitého kroku (v tomto případě se krokem rozumí výběr a aktualizace vhodné sady pixelů). Pokud bude daný krok zopakován několikrát, výstup bude hrubý a nesouvislý, další opakování by měla výstup postupně zdokonalovat až k ideálnímu obrazu.

Výše zmíněný krok byl poprvé představen v [1] jako tzv. *golden thread*. Podobně je v [2] *golden thread* definován jako proces, pomocí kterého je vybírána vhodná sada pixelů pro aktualizaci. Pokud je výběr této sady správný a její mohutnost optimální, aktualizace pixelů je dostatečně rychlá a výstup se bude postupně zdokonalovat. To se může projevit snížením prostorového rozptylu, k čemuž dochází právě ve statických scénách, nebo pokud se animace ve scéně zpomalí nebo úplně zastaví.

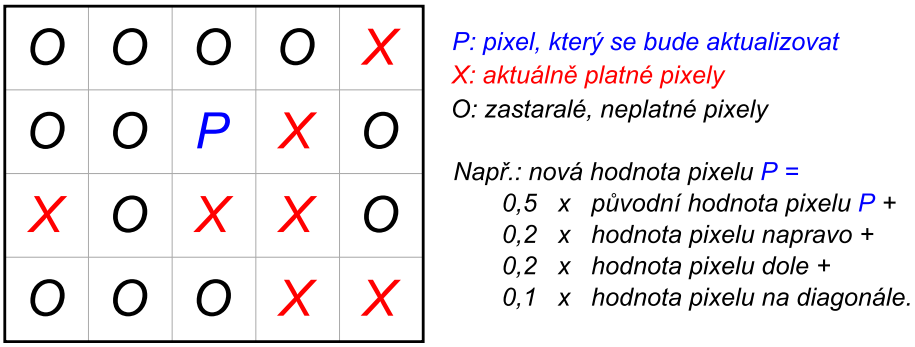
2.4.2 Výběr správné sady pixelů

Výběr správné sady pixelů vhodných k aktualizaci je důležitým krokem ke snížení prostorového rozptylu a tím i ke zlepšení celkové kvality výstupu. Způsobů, kterými definovat správnou sadu takových pixelů je více a v jednotlivých implementacích se rozcházejí. Např. v [2] byl zvolen náhodný výběr pixelů napříč rastrem a to tak, že v každém kroku je aktualizována vždy jedna třetina z celkového počtu všech pixelů. Tím, jak je v [2] uvedeno, je docíleno daleko plynulejších přechodů mezi jednotlivými snímky, než při klasickém způsobu využívajícím přítomnosti dvou bufferů. Jiným přístupem je využití řízeného vzorkování, které bylo představeno v [6] v rámci *adaptivní* varianty *bezsnímkového renderování* a je mu věnována kapitola 3.2. V [20] byly představeny tři možné přístupy, které řeší právě výběr sady pixelů podle informací dostupných přímo z rastru, ze scény nebo z okolních pixelů.

Hraniční vzorkování: jeden z uvedených způsobů spočívá v zaměření se na hrany objektů ve scéně. Scéna samotná si musí uchovávat informace o všech entitách, které obsahuje (světla, kameru, objekty – primitiva, materiály, textury, apod.) během animace. Aby bylo možné zaměřit se na hrany objektů, je nutné znát hodnotu gradientu ve všech pixelech. Tato hodnota bude vyšší právě v okolí hran. Porovnáním hodnot gradientu mezi sousedními pixely lze snadno odvodit, které z nich leží na nějaké hraně a při aktualizaci se zaměřit právě na ně.

Stárnutí pixelů podle jejich okolí: jelikož čistě náhodný výběr pixelů dosahuje většího výskytu prostorového rozptylu, je v [20] navrhnut post-processing obrazu využívající hodnoty sousedních pixelů ležících v jejich osmi-okolí. Hodnota každého pixelu, která již není platná (pixel nebyl po delší dobu aktualizován), může být upravena podle hodnot stále platných sousedních pixelů, kde pozice v osmi-okolí sousedního pixelu udává váhu, kterou je jeho hodnota vynásobena a přičtena k původní hodnotě neplatného pixelu. Váhy i samotný princip je znázorněn na obrázku 2.2. Sdílení informací mezi sousedními pixely lze chápat jako automatické zapojení konvolučního filtru, jehož výsledkem bude rozmazání obrazu a tím možné zlepšení kvality výstupu.

Energie pixelů je využita pro budoucí výpočty: Dalším konceptem, který je rovněž diskutován v [20], je možnost počítat s tzv. *energií pixelu*. Ta je definována jako cena nutná pro spočítání hodnoty daného pixelu a může být využita pro výpočet budoucích pixelů. Ty pak lze vybírat na základě predikce, která zohledňuje pohyb kamery, pohyby objektů a uživatelské vstupy. Výsledkem je pak redukce počtu pixelů, které je nezbytně nutné aktualizovat, což ovšem nemusí mít nutně vliv na snížení kvality obrazu.



Obrázek 2.2: Obrázek převzatý z [20] ilustruje využití vah novějších pixelů ležících v osmi-okolí neaktuálních pixelů. Pixel P je již neplatný. Proto je jeho hodnota před zobrazením upravena stále platnými sousedními pixely X, přičemž jejich hodnota je násobena vahou podle pozice v osmi-okolí a následně přičtena k původní hodnotě pixelu P.

2.4.3 *Bezsnímkový* vykreslovací řetězec

V [20] je popsán *vykreslovací řetězec* vykreslovací řetězec, který se skládá z *rendereru* a *displeje*. Na rozdíl od klasického chápaní displeje jako rastru pixelů zarovnaného do uniformní mřížky jsou diskutovány alternativy, které se více přibližují principům, na nichž je založena činnost lidského oka. *Ellen J. Scher Zaiger* v [20] definuje velikost jednoho vzorku přímo úměrně velikosti foto-receptorů v oku. To zdůvodňuje především procesy, které vedou k vyhodnocení barvy jako vjemu v lidském mozku. Přesněji, „výpočet“ barvy je založen na intenzitě a energii světla, které dopadá na sítnici v oku. Podle místa dopadu na sítnici je tato informace předána do mozku, kde je dále vyhodnocena.

Proto je navržen upravený *vykreslovací řetězec*, který více odráží principy lidského oka a mozku odehrávající se při rozpoznávání barev. V [20] je vykreslovací řetězec navržen následovně:

Renderer: Pro účely *bezsnímkového renderování* je navržen jako *renderer* klasický *ray-tracer*, který se dokáže vypořádat anti-aliasingem, umožňuje počítat osvětlení a stíny (tvrdé i měkké), difusní a spekulární odlesky a odrazy a je možné přizpůsobit velikost vzorku velikosti foto-receptoru v lidském oku.

Displej: Jak již bylo uvedeno v 2.1 a v 2.3, *bezsnímkové* renderování není principiálně stavěno na dnešní zobrazovací zařízení. Proto je nutné pro displej hledat alternativy založené spíše na vnímání s možností přizpůsobení velikosti vzorků. Distribuce vzorku na sítnici v lidském oku závisí na pozici oka a směru, kterým se uživatel dívá, reps. směru, ze kterého vzorek na sítnici dopadá, a místě kam dopadá. Proto *Ellen Zaiger*

v [23] navrhuje použití *head mounted* displejů. A aby byla informace úplná a nezkreslená, přiklání se k použití dvou nezávislých displejů, každého pro jedno lidské oko (k tomuto konceptu jsou nejblíže stereo kamery nebo právě *HMD*).

2.5 Faktory ovlivňující kvalitu výstupu

Metoda *bezsnímkového renderování* ve své základní podstatě, tak jak byla navrhnuta v [2], vede k výskytu *prostorového rozptylu* (z angl. *spatial scatter*), který je způsobený asynchronní aktualizací jednotlivých pixelů v rastru. To vede ke zhoršení kvality výstupu a může se projevovat v podobě „šumu“, neboť sousední pixely aktualizované v různých časových okamžicích mohou nést hodnotu, která odpovídá různým stavům scény, tedy různému rozložení primitiv ve scéně. Míra a hlavní faktory pro výskyt prostorového rozptylu byly popsány a formalizovány v [20] a mezi hlavní z nich patří:

- rychlost pohybu objektů a kamery ve scéně,
- počet a velikost objektů ve scéně a jejich vzájemná separace,
- složitost animací a pohybu objektů (počet pohybujících se objektů, konstantní rychlost versus zrychlení/zpomalení, vzájemné kolize objektů, apod.),
- celková složitost scény,
- vzdálenost objektů od kamery,
- ostrost a kontrast objektů na svých okrajích,
- osvětlení.

Pro formalizaci prostorového rozptylu bylo v [20] vycházeno především z teorie zpracování signálů. Díky ní lze popsat rušivé artefakty, které jsou vlastní metodě *bezsnímkového renderování*, a definovat je jako šum. Vzorkovací teorém (viz [15]) říká: „*rekonstrukce spojitého signálu z jeho vzorků je možná pouze tehdy, pokud byl vzorkován frekvencí alespoň dvakrát větší, než je maximální frekvence rekonstruovaného signálu, jinak dojde k aliasu.*“. Jelikož prostorový rozptyl obsahuje informace, které frekvence jsou zastoupeny a které ne, je možné získat signál pouze v určitém rozsahu (viz tzv. *scale space* teorie, [11]). A jelikož byly popsány funkce pro syntézu šumu jako signálu (např. v [9]), je možné tyto poznatky využít k popisu signálu, který je v prostorovém rozptylu obsažen. Díky tomu lze s prostorovým rozptylem nakládat jako se šumem a lépe ho formálně definovat, včetně faktorů, které se podílejí na jeho vzniku.

2.5.1 Formalizace prostorového rozptylu

Prostorový rozptyl vzniká asynchronní aktualizací jednotlivých pixelů v rastru a pokud je počet aktualizovaných pixelů příliš malý, nebo aktualizace neprobíhají dostatečně rychle, bude se rozptyl ve výstupu vyskytovat v podobě artefaktů. V [20] byla snaha vznik rozptylu a faktory podílející se na jeho vzniku formálně popsat. Tato podkapitola shrnuje diskutované závěry.

Pro jednoduchost se v dalším textu předpokládá, že velikost jednoho vzorku je rovna velikosti pixelu a výběr vzorků je realizován v náhodném pořadí podle [2].

V [20] je rozptyl definován pomocí *rychlosti*, *hustoty*, *plochy* a *gradientu*. Všechny tyto atributy přispívají k velikosti a množství vzniklého rozptylu. *Rychlost* se dále skládá ze *směru*, kterým se rozptyl pohybuje a *rozsahu* tohoto pohybu. *Rychlost* je definována pro každý pixel v rastru, je tedy dvourozměrná. Míra rozptylu σ přestavuje šířku jádra konvolučního filtru, který je schopen rozptyl rozmazáním původního obrázku odstranit:

$$I'(x, y) = G_\sigma \otimes I(x, y), \tag{2.1}$$

kde nový obrázek I' neobsahuje žádný šum do velikosti σ. Míru šumu lze pak získat jako maximum ze všech rozsahů pohybu v každém pixelu:

$$\sigma = \max(\sigma_{point_i}). \tag{2.2}$$

Hustota rozptylu je definována jako počet vstupů vzorkovaných za jednotku času. Hodnota šumu se rovná jedné právě tehdy, když je počet vstupů roven počtu pixelů, které lze aktualizovat za jednu vteřinu. To pak odpovídá klasickému způsobu využívajícího dvou bufferů. V tom případě je vzorkovací frekvence rovna celkovému počtu pixelů v rastru a prostorový rozptyl vůbec nevzniká. V podstatě se jedná o snahu o diskretizaci rozptylu na jednotlivé úrovně (od jemnějších k hrubším) a pokud jsou vstupy vzorkovány nepřetržitě, hustota rozptylu se přibližuje k jedné.

Plocha, kterou rozptyl pokrývá, zahrnuje všechny pixely obsahující alespoň jeden objekt. Okraje objektů tvoří *obrys rozptylu* (z angl. *scatter contour*). Ten ohraničuje nejmenší možnou oblast rozptylu pro jeden konkrétní objekt. V každém okamžiku pak *plocha* pokrývá takové množství pixelů, na kterých je po projekci zobrazen nějaký objekt. Obecně však nelze říci, že by celková plocha pokrytá rozptylem byla rovna součtu všech svých částí. Pokud S_{total} značí celkovou plochu a S_{obj_i} plochu, kterou pokrývá i-tý objekt a n je počet aktuálně viditelných objektů, je nutné předpokládat tuto nerovnost:

$$S_{total} \neq \sum_{i=1}^{n} S_{obj_i} \tag{2.3}$$

Posledním atributem je *gradient*. Ten se zpravidla používá pro získání lokální nebo globální informace o změně frekvence. Hodnota gradientu bývá vysoká v okolí hran a okrajů objektů, kdežto k jejich středům naopak klesá. V souvislosti s prostorovým rozptylem je gradient definován pomocí *scale-space* aparátu (viz [11]). Lze ho definovat právě pro jeden pixel, stejně tak pro jakýkoliv rozsah σ, reprezentující nejbližší okolí pixelu do vzdálenosti σ. Tuto vzdálenost lze pak použít jako velikost jádra konvolučního filtru pro rozmazání výstupu. V [20] je *gradient* definován jako:

$$\nabla F(x, y) = \begin{pmatrix} F_x \\ F_y \end{pmatrix}, \tag{2.4}$$

kde F je funkce popisující povrch objektu a F_x, resp. F_y, značí směrové derivace v x-ové, resp. y-ové, ose.

2.5.2 Pohyb objektů a kamery ve scéně

Ve snaze o formální definici rozptylu lze výše uvedené aplikovat na konkrétním případě. Z úvodu této kapitoly byl zvolen problém *rychlosti pohybu objektů a kamery ve scéně*. Pro ostatní problémy lze postupovat obdobně nebo lze vycházet z [20].

Rychlost pohybu objektů a kamery ve scéně je možné podrobně analyzovat, jelikož tyto jevy lze díky transformacím přesně matematicky popsat. A dokonce je lze sloučit do jediného problému. To je možné, neboť při pohybu kamery scénou dochází k transformaci pozice projekční roviny a vůči ní, resp. na ní, se transformují projektované, byť statické, objekty. Ty budou mít po projekci jiné pozice na projekční rovině než v předchozím okamžiku. Ve výsledku je tedy jedno, zda se pohybuje kamera vůči objektům, nebo objekty vůči projekční rovině. Výsledek bude prakticky totožný.

Jak již bylo řečeno, každý pohyb kamery způsobí změnu polohy projekční roviny a po projekci dojde k posunu všech objektů na projekční rovině. *Rychlost* rozptylu je pak přímo úměrná lineární a úhlové rychlosti, kterou se posouvá projekční rovina při pohybu kamery. Rozsah rozptylu, tak jak byl definován v kapitole 2.5.1, poté odpovídá vzdálenosti, kterou urazil bod projektovaný na projekční rovinu od své poslední aktualizace. Z toho vyplývá skutečnost, že směr rozptylu je dvourozměrný jednotkový vektor pohybu. Nechť $T = (T_x, T_y, T_z)$ značí lineární rychlost a R_θ rychlost úhlovou okolo obecné osy $A = (A_x, A_y, A_z)$. Pak $R = (R_x, R_y, R_z)$ je dekompozicí rotace okolo osy A úhlovou rychlostí R_θ. Bod P na projekční rovině bude posunut o ΔP následovně:

$$\Delta P_x = R_x P_y - R_y P_z - T_x \tag{2.5}$$
$$\Delta P_y = R_x P_z - R_x P_x - T_y \tag{2.6}$$
$$\Delta P_z = R_y P_x - R_x P_y - T_z \tag{2.7}$$

Ve dvourozměrném prostoru (na projekčním plátně, kde $\mathcal{R}^3 \to \mathcal{R}^2$, tedy $(x, y, z) \to (u, v)$) lze z komponent u a v bodu P vypočítat novou pozici bodu P' se složkami u' a v' v příštím časovém okamžiku po jeho posunu. Pak vektor $V = (u - u', v - v')$ udává rychlost rozptylu a délka vektoru $|V|$ udává jeho rozsah:

$$Vel_\sigma = |V| = \sqrt{(u - u')^2 + (v - v')^2} \tag{2.8}$$

Směr rozptylu je jednotkový (normalizovaný) vektor V:

$$Vel_{dir} = \frac{V}{|V|} = \frac{(u - u', v - v')}{\sqrt{(u - u')^2 + (v - v')^2}} \tag{2.9}$$

Z rovnic (2.8) a (2.9) byla odvozena základní veličina *rychlost* (z angl. *Velocity*) pro rozptyl vznikající v obraze. Pro analýzu pohybu objektů ve scéně je postup totožný (viz výše). V [20] jsou popsány i další faktory, které se podílejí na vzniku prostorového rozptylu a tedy i kvalitě výstupu — jejich výčet je v seznamu na začátku kapitoly 2.5 — avšak pro účely této práce je tento příklad dostačující.

2.6 Raytracer

Pro výpočet nové hodnoty – barvy – právě aktualizovaného pixelu je využito metody sledování paprsku pomocí *raytraceru*. Tato metoda je implementačně jednoduchá a pro aktualizaci pixelů napříč rastrem se přímo nabízí. Jelikož však není metoda *bezsnímkového renderování* metodou pro fotorealistické zobrazování, ale pro zobrazování počítačové grafiky v reálném čase, je nutné, aby použitý *raytracer* byl především rychlý. S přihlédnutím k algoritmu 2.2 je právě funkce `spočítejBarvuNa()` klíčová v rámci celkové odezvy aplikace. Z těchto důvodů není prioritou použití složitých *raytracerů* a pokročilých metod (měkké

stíny, stochastické odrazy paprsků, apod.), lze si vystačit pouze se základními funkcemi, které přesto nabízejí kvalitní výstup. Jedná se především o difuzi, odlesky, odrazy a lomy a tvrdé stíny.

2.6.1 Sledování paprsku a výpočet barvy

Výpočet nové barvy pixelu na konkrétní pozici v rastru je založen na vyslání primárního paprsku skrze projekční rovinu a jeho sledování ve scéně. Pokud je nalezen nejbližší průsečík mezi paprskem a primitivem ve scéně je výsledná hodnota pixelu složena ze tří pomocných barev. Pro výpočet základní barvy pixelu v bodě průsečíku s primitivem ve scéně lze využít tzv. *Phongův osvětlovací model*:

$$I_p = k_a + \sum_{m \in L} \left(k_d \left(\vec{L}_m \cdot \vec{N} \right) + k_s \left(\vec{R}_m \cdot \vec{V} \right)^{\alpha} \right), \tag{2.10}$$

kde k_a je *ambientní* složka, k_d je *difusní* složka a k_s složka *spekulární*. L představuje seznam všech nezastíněných světel ve scéně a $\vec{L}_m$ je vektor k m-tému světlu. $\vec{N}$ je normála v bodě dopadu paprsku a α určuje ostrost odlesků. $\vec{V}$ je pohledový vektor směřující do kamery a $\vec{R}_m$ je vektor odraženého paprsku:

$$\vec{R}_m = 2(\vec{L}_m \cdot \vec{N})\vec{N} - \vec{L}_m. \tag{2.11}$$

Rovnice (2.10) počítá základní barvu pixelu včetně její *difusní*, *spekulární* a *ambientní* složky podle aktuálního osvětlení, tedy podle pozice průsečíku v prostoru a všech aktuálně nezastíněných světel ve scéně. Pokud to vlastnosti materiálu objektu umožňují (materiály jako lesklý kov nebo zrcadlo), je spočítána barva odlesků v místě dopadu paprsku. To je realizováno pomocí rekurze vytvořením nového, *sekundárního paprsku* podle rovnice:

$$\vec{R}_{refl} = \vec{D} - 2(\vec{D} \cdot \vec{N})\vec{N}, \tag{2.12}$$

kde $\vec{D}$ je směr původního paprsku a $\vec{N}$ normála v bodě dopadu. Sekundární paprsek je rekurzivně odražen do scény z místa dopadu původního paprsku a proces se opakuje. Pokud je objekt v místě dopadu průhledný (materiál typu sklo nebo voda) je vytvořen opět nový, sekundární paprsek, který je zalomen podle rovnice (2.13), kde $\vec{N}$ je normála v bodě dopadu, $\vec{D}$ je směr původního paprsku a n_1 a n_2 indexy lomu v prvním a druhém prostředí. Nový paprsek je opět rekurzivně poslán do scény nebo dovnitř objektu a hledání pomocné barvy se opakuje.

$$\vec{R}_{refr} = -\cos\theta_1 \vec{N} + \frac{n_2}{n_1}(\cos\theta_2 \vec{N} + \vec{D}). \tag{2.13}$$

Rovnice (2.12) a (2.13) vytvářejí nový paprsek a využívají rekurzivního volání, které je prováděno do určité, explicitně nastavené hloubky. Výsledkem je vždy barva, která je přičtena k hlavní barvě pixelu z rovnice (2.10). Tím je získána kompletní nová barva pro aktualizovaný pixel.

2.6.2 Dělení prostoru scény – *kd*-strom

Spolu s metodou sledování paprsku je často nutné řešit i dělení scény pro rychlé nalezení průsečíku s nejbližším primitivem. Jednou z možností, které jsou pro tento účel určeny, je využití tzv. *kd*-stromu, který byl navržen a definován např.v [8] nebo v [19].

Konstrukce *kd*-stromu je realizována dělením prostoru podle jedné dělící roviny na dva podprostory. Dělení je prováděno rekurzivně do určité hloubky, kde každý podprostor reprezentovaný osově zarovnaným boxem (z angl. *axis aligned bounding box*) obsahuje buď další dva podprostory nebo primitiva, která se v daném boxu fyzicky nacházejí. Pro nalezení ideální dělící roviny a pozice na ní lze vycházet z heuristiky *SAH* (z angl. *surface area heuristic* definované např. v [18]. Ta hledá dělící pozici na jedné z rovin podle ceny každé možné dělící pozice. Dělící pozice jsou vybírány podle krajů primitiv a pozice s nejlepší cenou je pak zvolena pro rozdělení prostoru podle aktuální roviny na dva podprostory. Cena dělící pozice je počítána podle rovnice:

$$cost_{split}(V_L, N_L, V_R, N_R) = C_{trav} + P(B_L|V) \cdot N_L + P(B_R|V) \cdot N_R, \tag{2.14}$$

kde N_L, resp. N_R, je počet primitiv ležících v levém prostoru (boxu) B_L, resp. pravém prostoru B_R, a C_{trav} je vhodně zvolenou konstantou. Při dělení prostoru B na dva podprostory B_L a B_R je nejprve nutné určit pravděpodobnost $P(B_n|V)$ udávající, zda paprsek při průchodu stromem navštíví daný prostor. Pravděpodobnost pak lze počítat podle rovnice:

$$P(B_L|B) = \frac{SA(B_L)}{SA(B)} \quad \text{a} \quad P(B_R|B) = \frac{SA(B_R)}{SA(B)}, \tag{2.15}$$

kde $SA(B) = 2(B_xB_y + B_xB_z + B_yB_z)$ určuje plochu prostoru B s rozměry B_x, B_y a B_z v jednotlivých osách. Pro rozdělení každého uzlu stromu je proces volán rekurzivně na oba jeho pod-uzly. Koncovou hloubku rekurze, resp. výšku výsledného *kd*-stromu, je možné specifikovat explicitně nebo ponechat nastavení automatické, které je počítáno podle rovnice:

$$depth = \log_2 prim_{count}, \tag{2.16}$$

kde $prim_{count}$ udává celkový počet primitiv ve scéně. S využitím logaritmu o základu 2 bude hloubka stromu i pro mnoho primitiv relativně malá[5] a výsledný strom bude obsahovat přiměřené množství primitiv v každém listu. To se kladně projevuje na rychlosti hledání průsečíků s primitivy ležícími v jednotlivých listech *kd*-stromu.

2.7 Vhodný výběr vzorků pro aktualizaci

Bezsnímkové renderování, tak jak bylo navrženo v [2], se snaží snížit odezvu výstupu v grafických aplikacích výběrem a okamžitou aktualizací pixelů. Získání pixelu je prováděno čistě náhodným výběrem dvou čísel představujících pozici v x-ové a y-ové ose grafického rastru, např. pomocí nějakého generátoru pseudonáhodných čísel. Tento přístup je velmi rychlý a intuitivní. Stačí vybírat náhodné pozice vzorků na projekční rovině a doufat, že pokrytí obrazovky bude jak v čase tak v prostoru co nejlepší. Aby však *bezsnímkové renderování* generovalo přijatelný výstup, je žádoucí všechny pixely napříč projekční rovinou vybírat a) náhodně přes celou obrazovku, b) s přiměřeně stejnou četností ve všech jejích oblastech. To znamená, že by nemělo docházet k tomu, aby některé pixely byly aktualizovány častěji než jiné, nebo naopak, aby se na některé pixely úplně zapomnělo a neaktualizovaly se ani jednou. Jak bylo ukázáno v [14], náhodný výběr vzorků není úplně ideálním řešením a výstup získaný tímto přístupem obsahuje daleko více artefaktů.

[5] Pro 1 048 576 primitiv ve scéně bude hloubka *kd*-stromu 20.

Tato kapitola navrhuje alternativy pro výběr vhodných vzorků k jejich aktualizaci aplikováním jiných technik, než jaké byly původně v [2] navrženy. Konkrétně se jedná o tzv. *jittered* vzorkování (viz podkapitola 2.7.2), problém n věží (viz podkapitola 2.7.1) a *Poissonovy disky* (viz podkapitola 2.7.3). Na závěr kapitoly jsou představeny posloupnosti pseudonáhodných čísel a konkrétně *Haltonova posloupnost* (viz podkapitola 2.7.4), která byla vybrána jako ideální řešení v [14].

2.7.1 Problém n věží

Jedná se o klasický problém, který je analogií k šachové partii. Přesněji řečeno, popisuje, jak rozložit n věží na šachovnici o velikosti $n \times n$, tak, aby se vzájemně neohrožovaly. Každé políčko šachovnice představuje jednu buňku. Implementace tohoto problému je v celku snadná: nejprve se věže umístí na jednu z diagonál a následně se např. v x-ové ose provede jejich promíchání např. výměnou dvou věží na různých pozicích. Tuto činnost je nutné aplikovat několikrát po sobě. Výhoda tohoto postupu je ta, že věže představující jednotlivé vzorky jsou od sebe dostatečně daleko, tak, aby jejich celkový přínos byl rozprostřen přes celý zkoumaný region, v tomto případě celý obraz. Jediný případ, který může být problematický je ten, kdy by vzorky zůstaly na svém původním místě na diagonále.

Problém n věží je v rámci *bezsnímkového renderování* možné implementovat např. tak, že je nejprve určeno umístění všech věží v jednotlivých buňkách a poté jsou aktualizovány všechny pixely, ve kterých se věže vyskytují. Tím je zajištěna aktualizace n příštích vzorků. Problém je, že jedna buňka může pokrývat více pixelů a samo umístění věže v buňce pouze říká, že se tam bude příští vzorek vybírat. Pro jeho správnou pozici je proto nutné přičíst náhodný *offset* k levému hornímu rohu buňky a až tato výsledná hodnota bude určovat skutečnou pozici vzorku v rastru. Po aktualizaci n vzorků je nutné celý proces opakovat, včetně generování nového rozložení věží v buňkách.

2.7.2 *Jittered* vzorkování

Oblíbenější a též častěji používanější metoda např. při řešení *supersamplingu* u *raytraceru* je tzv. *jittered* vzorkování[6]. Obraz je rozdělen opět na $n \times n$ buněk[7], kde v každé buňce je potom náhodně zvolena pozice vzorku. Každý vzorek pak spadá právě do jedné buňky a vůči jejímu počátku je posunut o tento náhodný *offset*. Tato metoda může být tedy implementačně jednoduší a časově rychlejší než metoda n věží.

Pro potřeby *bezsnímkového renderování* je možné tuto metodu implementovat podobně jako problém n věží. V jednom kroku lze postupně generovat n^2 náhodných hodnot a ty přičítat k počátkům příslušných buněk. Vzniklá pozice pak udává skutečnou pozici vzorku v rastru. Tímto postupem bude zajištěno pokrytí obrazu pro příštích n^2 vzorků. Přesto, že je tato metoda velmi oblíbená, její využití v *bezsnímkovém rendereru* není úplně ideální a vybírané vzorky mají tendenci ke shlukování. Lze říci, že ze všech metod představených v této kapitole je výstup této metody nejhorší (viz obrázek 2.3) a je srovnatelný s náhodným výběrem vzorků (to záleží na kvalitě použitého generátoru pseudonáhodných čísel).

2.7.3 Poissonovy disky

Metoda *Poissonových disků* je opět rozšířená napříč implementacemi různých *raytracerů* a vzorky generuje vždy náhodně, avšak tak, aby neporušovaly jistou podmínku. Tato pod-

[6] Někdy označované též jako *stratified* vzorkování.
[7] Obecně jich může být $m \times n$.

mínka říká, že každý vzorek může od ostatních ležet v určité minimální vzdálenosti r_t, tedy každý vzorek má okolo sebe pomyslnou kružnici o poloměru r_t a v této kružnici se nesmí vyskytovat žádný jiný vzorek.

Tato metoda je vcelku vhodná pro zakomponování do *bezsnímkového rendereru*. Při implementaci je nutné počítat s tím, že žádný nový vzorek, splňující danou podmínku již nebude možné nalézt. Potom lze generovat n vzorků, které jsou postupně aktualizovány a pokud je tento počet dosažen, nebo není možné nalézt vyhovující pozici nového vzorku, staré vzorky jsou zapomenuty a výpočet začíná od začátku. Její časová složitost bude závislá především na n a na aktuálním počtu již vygenerovaných vzorků: nalezení nového vzorku náhodným výběrem jeho pozice a porovnání, zda neleží uvnitř kružnice jiného vzorku bude rychlejší pro menší počet vzorků.

2.7.4 Pseudonáhodné posloupnosti, *Haltonova* posloupnost

Zcela jiný přístup byl představen v [14]. Ten vychází z *kvazi-Monte Carlo* metod a představuje pseudonáhodné posloupnosti pro generování nových vzorků. Ty, jak je v [14] ukázáno, dávají daleko lepší výsledky, tedy lepší rozprostření vzorků v prostoru a čase. Přesto že je v [14] diskutováno, testováno a srovnáno hned několik posloupností, pro účely této práce bude představena pouze jediná – *Haltonova* posloupnost, která se ostatně ukazuje jako nejlepší (srovnání viz [14]). Aby však následující text dával smysl, je nutné definovat základní strukturu a operaci nad ní (viz [14] a [13]):

Definice 2.7.1. ***Posloupnost bodů:*** *posloupnost bodů P_n je množina $\{x_0, x_1, \ldots, x_{n-1}\}$ bodů $x_i \in \mathcal{I}^s = \langle 0,1)^s$, kde $s \in \mathbb{N}$, $n \in \mathbb{N}$ a platí:*

$$P_{n+1} = P_n \cup x_n.$$

Všechny body x_i v posloupnosti P_n leží v intervalu $\langle 0,1)^s$, kde s značí velikost prostoru. Pro účely bezsnímkového renderování *lze uvažovat $s = 2$, jelikož se bude vždy jednat o dvourozměrný rastr a každý bod x_i pak bude reprezentovat souřadnice daného vzorku v tomto rastru.*

Definice 2.7.2. ***Radikálně inverzní funkce (radical-inverse function):*** *nechť je $(a_j)_{j=0}^{\infty}$ reprezentací celého čísla $i \in \mathbb{N}_0$ o základu $b \geq 2$. Pak funkce Φ_b se nazývá* radikálně inverzní funkce *a je definována jako:*

$$\Phi_b : \mathbb{N}_0 \to \langle 0,1),$$

$$i = \sum_{j=0}^{\infty} a_j(i) b^j \mapsto \sum_{j=0}^{\infty} a_j(i) b^{-j-1}.$$

Funkce Φ_b funguje tak, že otáčí celočíselné hodnoty indexu i okolo desetinné čárky v číselné soustavě o základu b. Např. $(11)_{10} = (1011{,}0)_2 \to (0{,}1101)_2 = (0{,}8125)_{10}$.

Haltonovu posloupnost, která byla představena v [7], je pak možné definovat takto:

$$x_i^{Halton} = \left(\Phi_{b_1}(i), \ldots, \Phi_{b_s}(i)\right), \tag{2.17}$$

kde $b_1, \ldots, b_s$ jsou čísla, která je vhodné zvolit tak, aby tvořila posloupnost prvních s prvočísel. Pro případ *bezsnímkového renderování*, kde $s = 2$ je možné upravit rovnici (2.17) do tvaru:

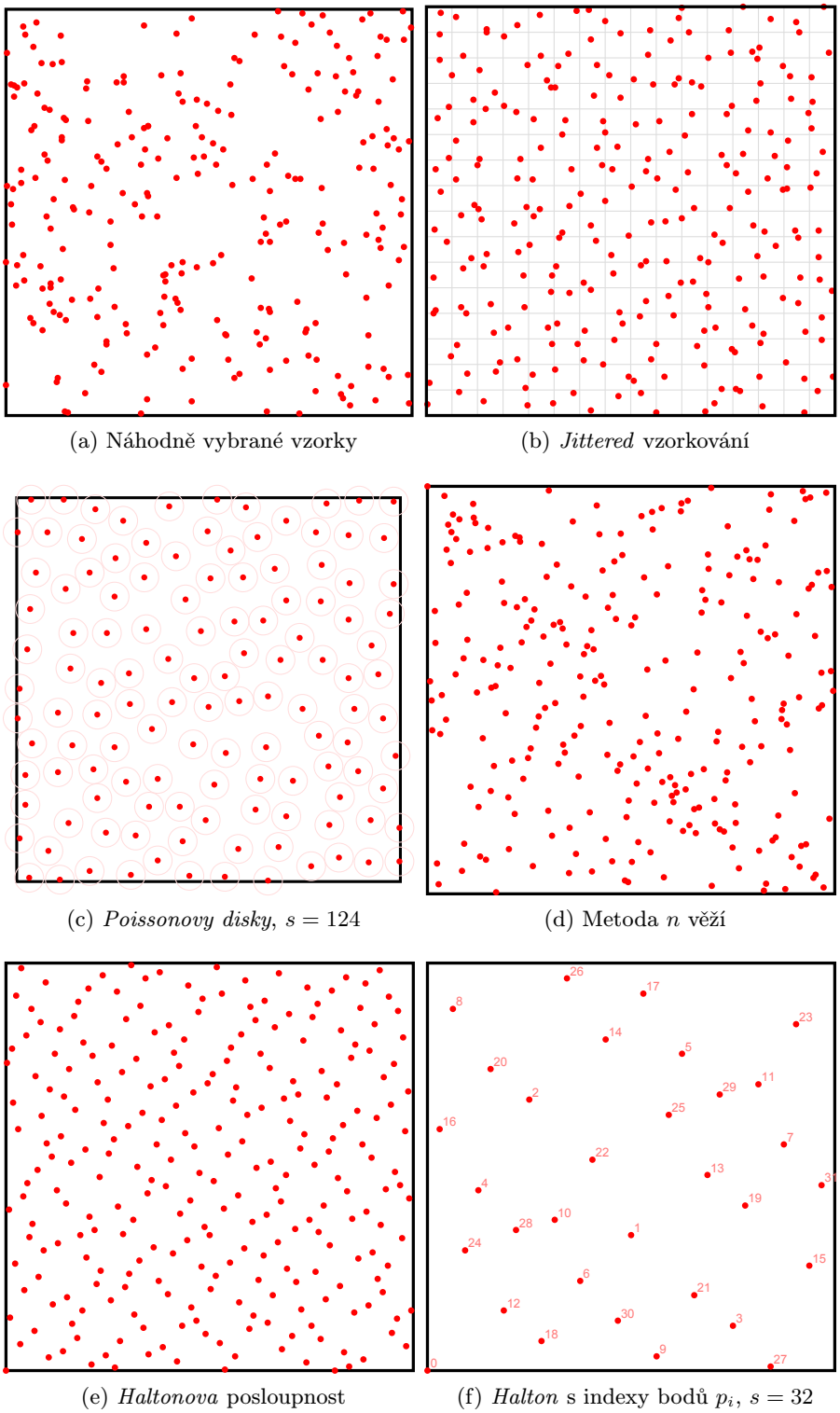
$$x_i^{Halton} = (\Phi_2(i), \Phi_3(i)), \tag{2.18}$$

kde $\Phi_2(i)$ reprezentuje x-ovou souřadnici a $\Phi_3(i)$ y-ovou souřadnici.

Jak již bylo řečeno, v [14] je představeno několik dalších posloupností, avšak právě *Haltonova* dosahuje nejlepších výsledků (viz obrázek 2.3) a v porovnání s ostatními vítězí. Navíc, její implementace je vcelku jednoduchá a je nastíněna pseudokódem 2.4.

Algoritmus 2.4: Pseudokód funkce pro výpočet jednoho prvku *Haltonovy* posloupnosti. Funkci je nutné volat dvakrát pro stejný `index` a jako proměnnou `base` zvolit nejprve číslo 2 (pro x-ovou složku) a následně 3 (pro y-ovou složku).

```
1  function HALTONPOINT(index, base)
2  begin
3      result ← 0;
4      factor ← 1 / base;
5      i ← index;
6      while i > 0 do
7          result ← result + factor * (i % base);
8          i ← floor(i / base);
9          factor ← factor / base;
10     end
11     return result;
12 end
```



(a) Náhodně vybrané vzorky (b) *Jittered* vzorkování

(c) *Poissonovy disky*, $s = 124$ (d) Metoda n věží

(e) *Haltonova* posloupnost (f) *Halton* s indexy bodů p_i, $s = 32$

Obrázek 2.3: Obrázek ilustruje činnost výše popsaných metod při výběru s vzorků ($s = 256$, pokud není uvedeno jinak). Zatímco náhodný výběr, problém n věží a *jittered* vzorkování (obrázky 2.3a, 2.3d a 2.3b) mají sklon ke shlukování vzorků, *Poissonovy disky* (obrázek 2.3c) a především *Haltonova* posloupnost (obrázek 2.3e) dokáží vzorky rozprostřít lépe přes celé plátno. Poslední obrázek ilustruje výběr prvních 32 vzorků *Haltonovou* posloupností včetně jejich pořadí v jakém byly vybírány.

Kapitola 3

Adaptivní metoda

Kapitola 2 se zaměřila na popsání a představení metody *bezsnímkového renderování* jako nového přístupu, kterým lze provádět zobrazování počítačové grafiky v reálném čase. Přesto, že jsme limitováni současnými technologiemi (grafické karty a zobrazovací zařízení jsou navrženy pro zobrazování celého výstupu najednou přepínáním mezi dvěma buffery), nemělo by nyní činit větší problémy implementovat vlastní *renderer*, který bude „*bezsnímkový*“, např. s využitím pseudokódu 2.3. Jak ale bylo uvedeno v kapitole 2, výstup bude obsahovat prostorový rozptyl v podobě artefaktů což bude mít nepříjemný vliv na jeho výslednou kvalitu.

Pro zlepšení výstupu *bezsnímkového* rendereru byla vyvinuta *adaptivní* metoda, která byla poprvé popsána a implementována v [6]. Na další implementaci pak lze narazit v [14]. Tato kapitola se zaměří na popis řízeného vzorkování a rekonstrukce obrazu, které představují její základní stavební bloky. Dále bude popsán samotný *adaptivní* algoritmus, tak jak byl představen v [6] a budou diskutovány různé přístupy návrhu a implementace z publikací uvedených výše.

3.1 Návrh *adaptivního* algoritmu

Podle definice metody *bezsnímkového renderování*, tak jak byla popsána v kapitole 2, je nový vzorek, který se bude aktualizovat, vybírán náhodně přes celý rastr. To však může způsobit (a také zpravidla způsobuje), že se budou aktualizovat pixely, jejichž změna není nutná, např. proto, že daná oblast obrazu je v nějakém delším časovém intervalu statická. A to na úkor jiné skupiny pixelů, kde právě probíhá nějaká složitá animace a aktualizace by zde tudíž měla probíhat častěji. Prvním krokem pro zlepšení výstupu *bezsnímkového rendereru* je upřednostnění výběru vzorků z oblasti obrazu tam, kde to má větší přínos. Tento postup lze nazvat *řízené vzorkování*[1] a je jedním ze dvou hlavních principů, které byly představeny s *adaptivní* metodou v [6].

Druhým principem, který byl rovněž představen v [6], je *rekonstrukce obrazu*. Bylo řečeno, že aktualizované pixely v klasickém pojetí *bezsnímkového renderování* jsou ihned zapisovány na výstup bez jakéhokoliv post-processingu. Ten je možný, je ale komplikovaný, jelikož aktuální výstup obsahuje pixely rozprostřené nejen v prostoru, ale i v čase. Aby bylo možné provádět rekonstrukci obrazu pomocí rekonstrukčních filtrů, je nutné brát v úvahu právě takové filtry, které zohledňují jak prostorovou, tak i časovou doménu.

[1] A angl. *guided sampling*.

Nyní je možné promítnout nové poznatky do současného algoritmu pro klasický *bezsnímkový renderer* (viz pseudokód 2.3) a ten převést na jeho *adaptivní* variantu. Nový algoritmus je nastíněn pomocí pseudokódu 3.1 a rozšiřuje původní algoritmus na dvou místech. Na řádku 4 je nejprve vybrán pixel z nějaké oblasti (funkcí `získejPixelVDůležitéOblasti()`), která je v aktuálním okamžiku nějak význačná a tedy důležitější než ostatní oblasti obrazu. Následuje výpočet barvy tohoto pixelu (např. pomocí *raytraceru*). Další změna oproti původnímu algoritmu je na řádku 6. `frameBuffer` není aktualizován pouze v rámci daného pixelu, ale provádí se kompletní rekonstrukce obrazu po přidání nové, právě spočítané, barvy funkcí `rekonstruujObrazPřidáním()`.

Algoritmus 3.1: Pseudokód pro *adaptivní bezsnímkové renderování*, rozšiřující algoritmus 2.3.

```
1 alokuj paměť pro frameBuffer o rozměrech X × Y;
2 forever do
3     while není nutné překreslit displej do
4         (x, y) ← získejPixelVDůležitéOblasti();
5         barva ← spočítejBarvuNa(x, y);
6         frameBuffer ← rekonstruujObrazPřidáním(barva);
7     end
8     prohoď oba buffery na grafické kartě a zobraz aktuální;
9 end
```

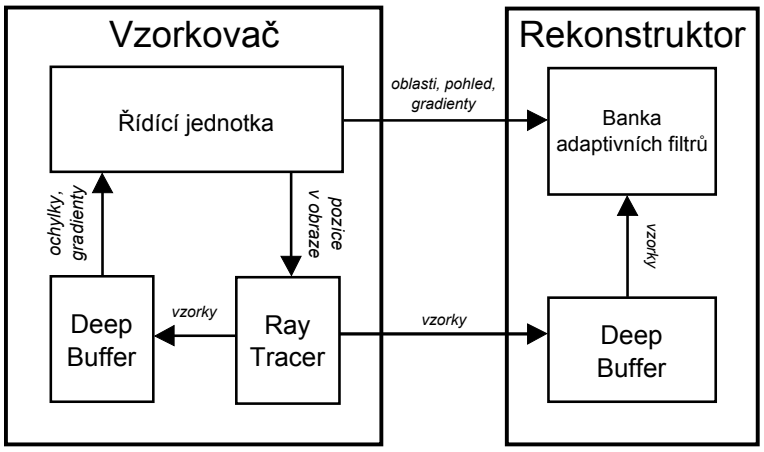
V [6] navrhují strukturu *adaptivního* algoritmu rozdělit do dvou nezávislých částí, kde první, *vzorkovač*, se stará o výběr vhodného vzorku ze správné oblasti (funkce `získejPixelVDůležitéOblasti()` z algoritmu 3.1) a o výpočet jeho nové hodnoty. Druhá komponenta pak provádí rekonstrukci výsledného obrazu (funkce `rekonstruujObrazPřidáním()`). Celý systém *adaptivního rendereru* je nastíněn na obrázku 3.1.

Vzorkovač (z angl. *sampler*) obsahuje řídící jednotku (*řadič*), *raytracer*, pro výpočet barev nových pixelů, a pomocný *deep buffer* pro ukládání, správu a výběr vhodných oblastí obrazu. Vzorkovač se snaží zvýšit vzorkovací frekvenci v těch oblastech obrazu, kde dochází k rozsáhlým změnám v rámci prostoru a času. Toho je dosaženo rozdělením celého rastru do menších oblastí, kde některé oblasti jsou upřednostňovány před ostatními. Díky tomu je možné vybírat ty vzorky, které mají pro celý obraz větší přínos.

Rekonstruktor se na druhou stranu stará o rekonstrukci obrazu ze všech vzorků uložených v jeho vlastním *deep bufferu*. Pokud je scéna statická, dokáže poskytnout relativně kvalitní výstup bez aliasů, pokud se scéna mění, výsledky mohou být rozmazané, jsou ale poskytovány s dostatečně nízkou odezvou. To je prováděno pomocí filtrů, které jsou udržovány a aplikovány v *bance adaptivních filtrů*. Po každém hotovém vzorku si rekonstruktor od vzorkovače vyžádá nový vzorek a další pomocné informace a proces je opakován.

3.2 Řízené vzorkování

Cílem *bezsnímkového renderování* je aktualizovat pixely napříč projekčním plátnem co nejvhodněji. Tedy v nějakém náhodném pořadí, avšak tak, aby žádný pixel nebyl aktualizován



Obrázek 3.1: Komponenty *adaptivního bezsnímkového rendereru* a tok dat mezi nimi tak, jak byl navržen a popsán v [6] odkud byl převzat i tento obrázek.

častěji než ostatní, jinými slovy, aby pokrytí aktualizovaných pixelů bylo rovnoměrné a to jak v prostoru, tak i v čase. Co když ovšem bude v nějaké konkrétní situaci nějaká část obrazu po delší dobu statická a naopak jiná část obrazu bude procházet nějakou výraznou změnou? Klasickým přístupem v *bezsnímkovém renderování* by byly pixely vybírány náhodně, tedy zhruba stejně ve statické i dynamické oblasti plátna. To sice není úplně špatné a jistě to neodporuje žádným definicím, výsledek na druhou stranu nebude úplně ideální.

Bylo by lepší výběr nového vzorku napříč obrazem ovlivnit tak, aby byl vybrán s větší pravděpodobností právě z oblasti, kde dochází k nějaké větší změně, než z oblasti, kde je scéna statická. Tedy nějakým vhodným způsobem řídit výběr nového vzorku a obraz rozdělit do oblastí s různou prioritou. Oblasti s vyšší prioritou by byly právě ty, kde dochází ke změnám a byly by upřednostněny pro výběr nového vzorku.

3.2.1 Dělení obrazu

Dělení obrazu[2] lze provádět mnoha způsoby, avšak pro účely této práce, jakož i *bezsnímkového renderování* bude dělení obrazu, v tomto případě projektivní roviny, definováno následovně: projektivní rovina P_t o rozměrech $M \times N$ je rozděleno v čase t:

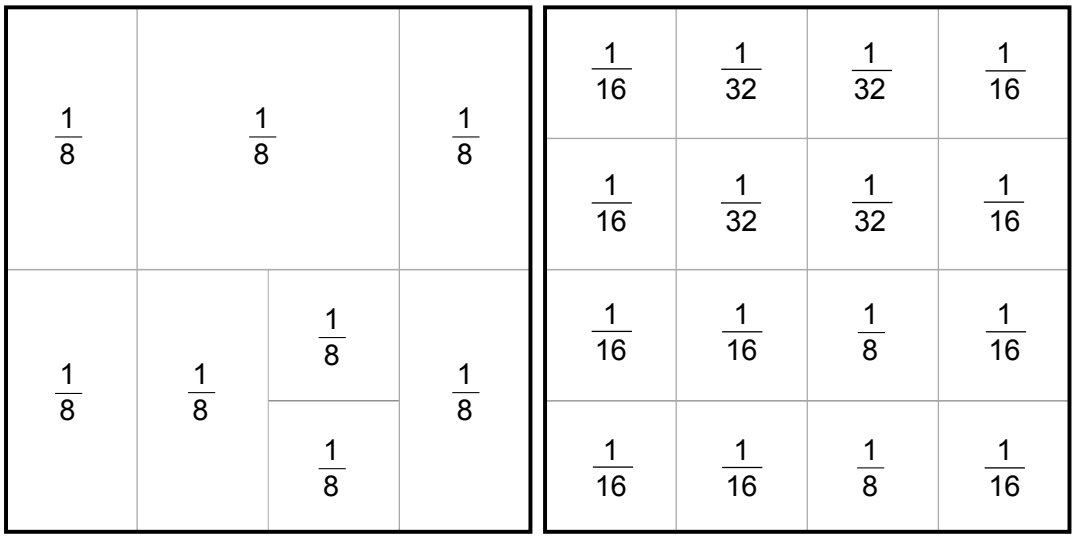
$$P_t = \{i(x, y, t_l) | (x, y) \in M \times N\}, \tag{3.1}$$

na n disjunktních oblastí $U_{i,t} \subseteq P_t$:

$$P_t = \bigcup_{i=1}^{n} U_{i,t} \quad \text{kde} \quad U_{i,t} \cap U_{j,t} = \emptyset. \tag{3.2}$$

$i(x, y, t_l)$ značí barvu pixelu v plátně na pozici (x, y) v čase jeho poslední změny t_l. Aby bylo možné celý obraz rozdělit na disjunktní množiny, tj. nepřekrývající se *oblasti*, a zároveň všemi oblastmi pokrýt všechny pixely obrazu, je rozumné definovat *oblasti* jako čtvercové, resp. obdélníkové. K této myšlence pro dělení projektivní roviny přistupují i v [6] a v [14] a oblasti $U_{i,t}$ definují jako:

[2] Dělení obrazu vychází z anglického *tiling* a jedna oblast obrazu se pak v anglické literatuře nazývá *tile*. Pro účely této práce budou používány české ekvivalenty.



(a) Stejná pravděpodobnost, různá plocha (b) Stejná plocha, různá pravděpodobnost

Obrázek 3.2: Dva způsoby dělení obrazu na *oblasti* diskutované v této kapitole. V [6] je navrhováno dělení obrazu na různě veliké oblasti, přičemž všechny pak mají stejnou prioritu výběru (obrázek 3.2a). Naproti tomu je v [14] navržen opačný přístup demonstrovaný na obrázku 3.2b. Všechny *oblasti* mají stejné rozměry a tedy i plochu, ale různou pravděpodobnost výběru vzorku. V tomto příkladu je více pravděpodobné, že příští vzorek bude vybrán z oblasti dole vpravo než z oblasti nahoře uprostřed.

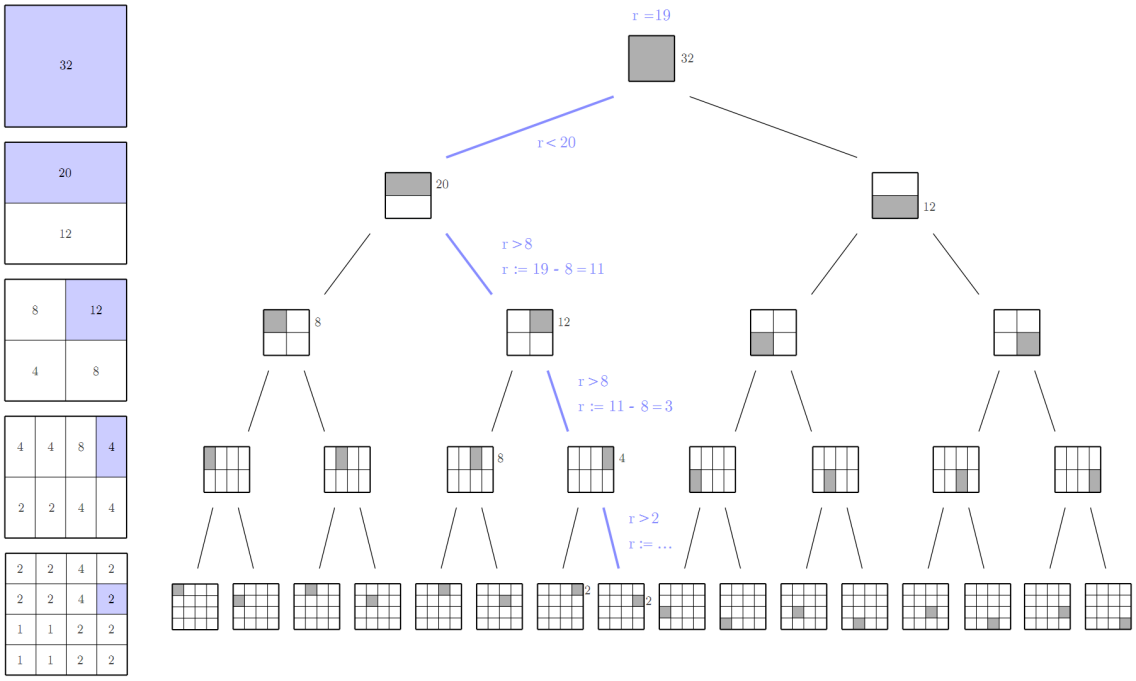
$$U_{i,t} = \{i(x, y, t_l) | (x, y) \in A \}, \tag{3.3}$$

kde $A = (u, \ldots, u + m) \times (v, \ldots, v + n)$, $1 \leq u \leq M$, $1 \leq v \leq N$, $0 \leq m < M$, $0 \leq n < N$ a tedy $u + m \leq M$ a $v + n \leq N$.

Jelikož byla *oblast obrazu* definována i v časové doméně (index t u každé oblasti U_i), bude nutné vyřešit problém, ve kterém okamžiku vytvořit, nebo dokonce změnit, uspořádání jednotlivých *oblastí* na projektivní rovině. Jelikož z výše uvedených rovnic nevyplývají žádná omezení pro rozměry oblastí, je možné, aby se jednotlivé oblasti lišily v čase svou velikostí a tedy i plochou. Dále lze předpokládat, že každá *oblast* ponese informaci o tom, jak je aktuálně významná, resp. předpokládejme, že existuje funkce $h(U_{i,t})$, která bude aktuální *prioritu* oblasti $U_{i,t}$ poskytovat. Její detailní popis bude následovat v podkapitole 3.2.2.

Pomocí rovnice (3.3) je možné obraz rozdělit dvěma možnými způsoby, tak jak to je nastíněno na obrázku 3.2. První způsob předpokládá rozdělení obrazu na oblasti různých rozměrů, kde všechny oblasti mají stejné ohodnocení (viz obrázek 3.2a). Naopak druhý způsob dělí obraz na oblasti stejných rozměrů, přičemž jejich ohodnocení se může lišit (viz obrázek 3.2b). Oba způsoby přinášejí různé výhody a nevýhody.

První způsob je implementován v [6] a má výhodu tu, že *oblast*, ze které se bude vybírat nový vzorek, je možné vybrat náhodně. Tedy pokud jsou všechny oblasti uloženy např. v poli, lze vygenerovat náhodný index do tohoto pole a jako vhodnou oblast zvolit právě tu, která na daném indexu leží. Vybráním oblasti se změní její ohodnocení a jelikož všechny oblasti musí mít ohodnocení stejné, je nutné obraz rozdělit znovu. To je ale časově velmi náročná operace, která bude mít přímý dopad na odezvu celé aplikace (např. v [6] to obcházejí tím, že obraz dělí po každých 25 zpracovaných vzorcích). Dalším problémem,



Obrázek 3.3: Obrázek převzatý z [14], který ilustruje binární strom jako strukturu vhodnou pro hledání *oblasti* U_i z níž se bude vybírat nový vzorek pro aktualizaci podle obrázku 3.2b. V tomto příkladu je plátno rovnoměrně rozděleno na 16 oblastí, které jsou uloženy v listech stromu. Každý uzel N_j udržuje sumu ohodnocení všech svých pod-uzlů a ta je propagována až ke kořeni (viz levý sloupec). Strom zobrazený vpravo demonstruje výběr oblasti U_i podle náhodně vygenerovaného čísla $r = 19$, které pomocí rovnice (3.4) postupně prochází stromem od kořene k listům. Pokud je r menší než ohodnocení levého pod-uzlu, je vybrána tato cesta, v opačném případě je toto ohodnocení odečteno od r a porovnáno s pravým pod-uzlem. Výsledná cesta je znázorněna modře.

který přináší tento způsob, je zvolení ideální váhy pro všechny oblasti, tedy do jaké hloubky obraz dělit.

Druhý způsob je naopak preferován v [14] a funguje na principu rozdělení obrazu na uniformní oblasti (např. 8×8 oblastí). Jelikož jsou rozměry všech oblastí stejné, je možné obraz rozdělit pouze jednou, ideálně před začátkem běhu aplikace. Díky tomu je možné úplně odstranit závislost jednotlivých oblastí $U_{i,t}$ na čase a nadále uvažovat oblasti pouze jako U_i. Nevýhodou může být na první pohled výběr oblasti, jelikož nyní není možné oblast vybrat náhodně, neboť každá má jiné ohodnocení. V [14] je navržena struktura pro uložení oblastí v binárním stromě, tak, že v listech jsou uloženy samotné oblasti se svým ohodnocením (viz obrázek 3.3). Každý uzel pak obsahuje sumu ohodnocení všech svých potomků, tedy ohodnocení $s_{N_j} = \sum_{c=1}^{k} s_{N_c}$, kde N_j je nějaký uzel stromu, k je počet jeho potomků a s_{N_c} je suma c-tého potomka uzlu N_j. Samotný výběr oblasti pak lze implementovat např. vygenerováním náhodného čísla r, které je v rozsahu od 0 do sumy všech ohodnocení (ohodnocení kořenového uzlu), tedy $r \in \langle 0, \sum_{i=1}^{n} h(U_i)\rangle$. Tato náhodná hodnota se pak nechá „propadnout“ stromem od kořene až k listům. Pro tento účel je v [14] navržena rekurzivní funkce δ, která vrací uzel N_b s konkrétní oblastí $b = \delta(r, 1)$:

$$\delta(r,c) = \begin{cases} c & \text{pokud} \quad r < s_{N_c} \quad \text{nebo} \quad c = k, \\ \delta(r - s_{N_c}, c+1) & \text{jinak,} \end{cases} \tag{3.4}$$

kde c je index c-tého potomka uzlu N_j. Pokud je b spočítáno pro všechny uzly, lze spojit rovnici (3.4) s výběrem náhodného čísla r a sestrojit algoritmus τ, který přijímá náhodné číslo r a kořenový uzel N_r a vrací index vybrané oblasti i:

$$\tau(r, N_j) = \begin{cases} i & \text{pokud} \quad N_j = U_i \quad \text{je listem,} \\ \tau(r - \sum_{c=1}^{b-1} s_{N_c}, N_b) & \text{jinak.} \end{cases} \tag{3.5}$$

Ačkoliv se může zdát tento přístup na první pohled komplikovaný, nabízí celkem intuitivní postup pro výběr vhodné *oblasti* a co víc, pozice ani plocha oblastí se v čase t nemění. Po výběru nového vzorku v nalezené oblasti je změna ohodnocení oblasti pomocí výše zmíněné — avšak zatím nedefinované — funkce $h(U_i)$ velmi jednoduchá. Jelikož všechny uzly nesou sumu ohodnocení svých potomků, nové ohodnocení stačí propagovat od listu s vybranou oblastí až po kořenový uzel stromu. Tento druhý způsob dělení obrazu se jeví jako efektivnější, jednodušší a hlavně rychlejší pro samotný běh aplikace.

Na závěr je nutné poznamenat, že aplikací algoritmu τ z (3.5) není získán přímo vzorek vhodný pro aktualizaci, ale pouze jedna konkrétní *oblast*, ze které se bude nový vzorek teprve vybírat. Konkrétní pozici vzorku lze získat např. náhodným výběrem souřadnic (x_r, y_r) (za předpokladu, že jsou souřadnice vybírány z intervalu $\langle 0, 1)$) nebo pomocí technik diskutovaných v kapitole 2.7. Novou pozici pak vynásobit relativní velikostí *oblasti* U_i a tím získat hodnotu posunu $(\Delta x, \Delta y)$ od počátku (x_{U_i}, y_{U_i}) oblasti U_i. Jejich součet pak reprezentuje skutečnou pozici (x, y) vzorku v rastru, který je nyní možné aktualizovat.

3.2.2 Heuristika $h(U_i)$

Posledním krokem v rámci *řízeného vzorkování* je definovat váhovou funkci $h(U_i)$ jejíž existence byla nastíněna v podkapitole 3.2.1. Funkce $h(U_i)$ by měla umět rozhodnout, které oblasti obrazu U_i mají být upřednostněny pro výběr nového vzorku před jinými. Způsobů, jak tuto funkci navrhnout a implementovat je více, pro účely této práce bylo vycházeno z [14], kde je funkce $h(U_i)$ definována jako druhá mocnina střední odchylky na množině hodnot S_j, tedy:

$$\sigma_n^2 = \frac{1}{n} \sum_{j=1}^{n} (S_j - \overline{S})^2, \quad \text{kde} \quad \overline{S} = \frac{1}{n} \sum_{j=1}^{n} S_j. \tag{3.6}$$

Za předpokladu, že σ značí váhu *oblasti* U_i, lze rovnici (3.6) pro případ *bezsnímkového renderování* upravit do tvaru:

$$\sigma^2(U_i) = \frac{1}{|A|} \sum_{(x,y) \in A} \left[i(x, y, t_l) - \overline{i} \right]^2, \quad \text{kde} \quad \overline{i} = \frac{1}{|A|} \sum_{(x,y) \in A} i(x, y, t_l). \tag{3.7}$$

Problém je, že rovnice (3.7) se zabývá pouze *statickým* signálem, což je pro účely *bezsnímkového renderování* nedostačující. Jelikož je však možné v *bezsnímkovém renderování* zpracovat pouze jeden vzorek v konkrétním okamžiku, je jasné, že novější vzorky budou mít vždy větší přínos pro určení ohodnocení oblasti U_i než ty starší. Proto je nutné do rovnice (3.7) přidat váhovou funkci λ, která bude aplikována na každý vzorek $i(x, y, t_l)$ a bude upravovat jeho hodnotu podle jeho časového razítka t_l:

$$\sigma^2(U_i) = \frac{1}{w} \sum_{(x,y)\in A} \lambda\left(i(x,y,t_l)\right) \cdot \left[i(x,y,t_l) - \bar{i}\right]^2, \tag{3.8}$$

kde

$$\bar{i} = \frac{1}{w} \sum_{(x,y)\in A} \lambda\left(i(x,y,t_l)\right) \cdot i(x,y,t_l) \quad \text{a} \quad w = \left(\sum_{(x,y)\in A} \lambda\left(i(x,y,t_l)\right) \right).$$

Bohužel i při aplikaci rovnice (3.8) může nastat problém. Pokud je nějaká část projektivní roviny delší dobu stejnorodá, tedy pokud obsahuje například jednu barvu, je možné, aby $\sigma^2(U_i) = 0$. To by ale mělo za následek, že by tato oblast v budoucnu již nikdy nebyla pro výběr nového vzorku upřednostněna. V [14] je proto navržena alternativa:

$$\sigma'(U_i) = \begin{cases} t_s & \text{pokud} \quad \sigma^2(U_i) \leq t_s, \\ \sigma^2(U_i) & \text{jinak}, \end{cases} \tag{3.9}$$

kde t_s představuje vhodně[3] zvolený práh (tzv. *starvation threshold*), jehož hodnota je vrácena v případě, že výsledek funkce σ^2 je menší než tento práh. To zajistí, že žádná oblast nebude mít nulovou pravděpodobnost a bude možné ji někdy v budoucnu znovu zvolit pro výběr nového vzorku.

Zbývá definovat funkci λ z rovnice (3.8). Jedná se o váhovou funkci, která upraví hodnotu aktuálního vzorku podle jeho stáří (viz výše). Jak bylo řečeno v kapitole 2.4, resp. jak je navrženo v [21], foto-receptory v lidském oku jsou citlivější na nejnovější informace a rychle se utlumují. Z toho plyne, že by bylo vhodné dávat největší váhu nejnovějším vzorkům a vzorky starší více než např. půl vteřiny by výběr oblasti neměly ovlivnit. Aby bylo možné přidělit největší váhu nejmladším vzorkům, je vhodné zvolit funkci λ jako inverzní exponenciální funkci. V [6] definují funkci λ jednoduše jako $\lambda = e^{-3,47a}$, kde a představuje vzrůstající věk vzorků v dané oblasti. V [14] je přístup podobný:

$$\lambda\left(i\ \ (x,y,t_l)\right) = e^{(t_l - t_a)/s}, \tag{3.10}$$

kde t_a značí čas nejnovějšího vzorku vybraného z oblasti U_i (tedy prakticky aktuální čas) a s je faktor, kterým lze ovlivňovat celkový přínos váhové funkce λ. Jelikož t_l označuje čas poslední aktualizace pixelu (x, y), je jasné, že vždy bude platit nerovnost $t_l - t_a \leq 0$ a tedy i $\lambda(i(x,y,t_l)) \leq 1$.

Váhová funkce λ pracuje s barvou pixelu (x, y) v čase t_l. Barva je v počítačové grafice zpravidla reprezentována pomocí tří složek R, G a B. Zbývá tedy definovat převod barvy na jednu hodnotu – *prioritu oblasti*. Možností jak toho dosáhnout je více, např. vzít průměrnou hodnotu ze všech tří komponent: $(R + G + B)/3$. Jako výhodnější přístup postačí využití informace o intenzitě barvy, tedy převedení dané barvy $i(x, y, t_l)$ do odstínů šedi pomocí rovnice:

$$Y = 0{,}299 \cdot R + 0{,}587 \cdot G + 0{,}114 \cdot B. \tag{3.11}$$

[3] Např. v [14] navrhují hodnotu 0,01.

3.2.3 Algoritmizace heuristiky $h(U_i)$

Rovnice (3.8) pro výpočet pravděpodobnosti pro výběr oblasti U_i dosahuje časové složitosti $\mathcal{O}(|A|)$, resp. $\mathcal{O}(m{\cdot}n)$ při dodržení notace z rovnice (3.3), kde m značí počet pixelů pokrytých oblastí v horizontálním směru a n počet pixelů ve vertikálním směru. To samo o sobě představuje značné zpomalení aplikace a jelikož je nutné volat funkci $\sigma'(U_i)$ po každém zpracovaném vzorku, bude zvýšení odezvy ještě umocněno. Např. pro rozlišení obrazu 256×256 a pro počet oblastí 8×8 bude nutné v každém volání funkce $\sigma'(U_i)$ iterovat nad 1 024 pixely. A to jen pro zajištění správného výběru oblasti v příštím kroku.

Z toho důvodu je v [14] představena inkrementální verze rovnice (3.8), která snižuje časovou složitost až na $\mathcal{O}(1)$. To přináší výrazné urychlení výpočtu pravděpodobnosti a tím i celého řízeného vzorkování, které *bezsnímkovou* metodu více nezatěžuje. Pro snížení časové složitosti je nutné nalézt inkrementální verze obou sčítanců a počítat jejich odchylky mezi kroky a a $a+1$. Nejprve je nutné vyjádřit očekávanou hodnotu $\bar{i}_a$:

$$\bar{i}_a = \frac{1}{w_a} \sum_{l=1}^{a} e^{(t_l - t_a)/s} \cdot i_l, \quad \text{kde} \quad w_a = \sum_{l=1}^{a} e^{(t_l - t_a)/s}, \tag{3.12}$$

kde $a = |A|$ a $i(x, y, t_l) = i_l$ (pro zjednodušení). Do této rovnice byla zahrnuta i váhová funkce λ. Tímto způsobem je možné přičítat barvu každého nového vzorku z oblasti U_i a vážit ji podle rozdílu v čase t_a (váha v aktuálním čase) a čase t_l (suma všech vah). Klíčový krok je nyní spočítat očekávanou hodnotu $\bar{i}_{a+1}$, která závisí pouze na $\bar{i}_a$:

$$\bar{i}_{a+1} = \frac{1}{1 + e^{(t_a - t_{a+1})/s} \cdot w_a} \left[i_{a+1} + e^{(t_a - t_{a+1})/s} \cdot w_a \cdot \bar{i}_a \right]. \tag{3.13}$$

Pro samotný výpočet je zapotřebí ukládat pouze poslední čas vzorku v dané oblasti t_a a poslední celkovou váhu w_a. Obě hodnoty je nutné aktualizovat vždy po přidání nového vzorku nastavením $t_a = t_{a+1}$ a $w_a = w_{a+1} = 1 + e^{(t_a - t_{a+1})/s} \cdot w_a$. Kromě očekávané hodnoty $\bar{i}$ je nutné počítat i druhou mocninu očekávané hodnoty $\overline{i^2}$, kterou lze získat ve své inkrementální podobě obdobně:

$$\overline{i^2}_{a+1} = \frac{1}{1 + e^{(t_a - t_{a+1})/s} \cdot w_a} \left[i_{a+1}^2 + e^{(t_a - t_{a+1})/s} \cdot w_a \cdot \overline{i^2}_a \right]. \tag{3.14}$$

Výslednou hodnotu pravděpodobnosti pro danou oblast $\sigma^2(U_{i,a+1})$, kde a je počet vzorků v dané oblasti, lze počítat z předchozí pravděpodobnosti $\sigma^2(U_{i,a})$ podle:

$$\sigma^2(U_{i,a+1}) = \overline{i^2}_{a+1} - \bar{i}_{a+1}. \tag{3.15}$$

3.3 Rekonstrukce obrazu

Druhou komponentou *adaptivní* metody — tak jak je popsána v [6] a přiblížena v kapitole 3.1 — je *rekonstruktor*. Ten se, jak již z názvu vyplývá, stará o rekonstrukci obrazu, která je klíčovou součástí v *adaptivním bezsnímkovém renderování*. Právě díky rekonstrukci obrazu vypadá výstup adaptivní metody lépe, než při použití původního konceptu z [2] (srovnání viz obrázek 2.1). Rekonstrukce obrazu je realizována pomocí *rekonstrukčního filtru*. Avšak v souvislosti s *bezsnímkovým renderováním* je nutné brát v potaz nejen tři dimenze — dvě prostorové a jednu časovou — ale dimenze čtyři. Tou poslední je tzv. *míra lokální vzorkovací hustoty*, která říká, že pokud je nějaká *oblast obrazu* vzorkována častěji než jiné oblasti, vliv jednoho vzorku by měl být menší než v jiných oblastech, které jsou

vybírány pro vzorkování zřídka. A dále, pokud je nějaký vzorek mladší než ostatní vzorky v nějakém regionu, jeho hodnota bude mít větší vliv na hodnotu jeho starších sousedů.

V této kapitole bude nastíněn výběr vhodného *rekonstrukčního filtru* r, rozbor jeho tvaru a velikosti a zvolení vhodné funkce, která filtr reprezentuje.

3.3.1 Určení tvaru a rozměrů filtru

Jak je ukázáno v [6] výběr vhodného filtru pro *adaptivní bezsnímkové renderování* není triviální záležitostí a přináší sebou problémy, které vyplývají z podstaty adaptivní metody. Je nutné specifikovat vlastnosti, které by měl ideální filtr mít a jejich adaptaci v závislosti na změně *lokální vzorkovací hustoty*. V [6] navrhují, aby byl filtr:

- široký v prostorové doméně a úzký v časové doméně pokud jsou změny v nějaké oblasti obrazu výrazné a naopak,
- aby byl úzký v prostorové doméně a široký v časové doméně pokud je nějaká oblast delší dobu statická.

Tyto dva návrhy mají jistě své opodstatnění: a) pokud se oblast mění dramaticky, je nutné zachovat co nejrychlejší odezvu i za cenu horší kvality výstupu. V důsledku pak jednotlivé vzorky rychleji stárnou. A za b) pokud je nějaká oblast delší dobu statická, hodnota starších vzorků bude stále relevantní, tj. bude stále odrážet skutečnost. Jak je patrné, jde o dva protipóly, které mohou nastat i v rámci jednoho obrázku. Proto není možné zvolit jeden statický filtr, ale je nutné tvar a rozměry filtru přizpůsobovat daným okolnostem průběžně.

Vhodným kritériem pro přizpůsobení filtru je využití hodnoty gradientu v dané oblasti. Jak je v [6] naznačeno, vysoká hodnota *prostorového* gradientu signalizuje přítomnost skutečné hrany v obraze (hrana objektu apod.), naopak vysoká hodnota *časového* gradientu signalizuje přítomnost dočasné hrany (stín, apod.).

Nechť je f_I obrazovou funkcí. Jelikož není nutné zachytit změnu gradientu ve všech třech dimenzích naráz, je v [14] navrhnuto, aby výpočet probíhal ve směru osy x, y a t odděleně. Hodnotu gradientu pak lze zjednodušit na absolutní hodnotu parciálních derivací v příslušných směrech, tedy:

$$d_x = \left|\frac{\partial f_I}{\partial x}\right|, \quad d_y = \left|\frac{\partial f_I}{\partial y}\right| \quad \text{a} \quad d_t = \left|\frac{\partial f_I}{\partial t}\right| \tag{3.16}$$

a tyto derivace převést na „*časoprostorový*“ objem V_s, který pokrývá prostor jednoho vzorku ve všech třech dimenzích. Objem V_s však lze vyjádřit i pomocí vzorkovací frekvence nějaké konkrétní oblasti v obraze, tedy „*kolik vzorků je zapotřebí k pokrytí určitého prostoru v čase*“. Nechť funkce $R(U_i)$ vrací počet vzorků na jeden pixel v nějakém časovém intervalu pro nějakou oblast U_i. Pokud vzorek s patří do oblasti U_i, objem V_s lze definovat jako:

$$V_s = \frac{1}{R(U_i)}. \tag{3.17}$$

Jak bylo řečeno v kapitole 3.2.1, každá oblast obrazu U_i pokrývá právě $m \times n$ pixelů, kde $1 \leq m \leq M$ a $1 \leq n \leq N$. Pro získání počtu vzorků, by tedy stačilo sečíst všechny vzorky v U_i, které byly vypočteny v určitém časovém intervalu. Tento interval lze definovat jako $\langle t_h, t \rangle$, kde t_h označuje poslední čas, kdy byl vzorek platný a t čas aktuální. Jak ale

zjistit poslední čas, kdy byl ještě vzorek platný? V [6] např. uvažují počet vzorků za jednu vteřinu, tedy $t_h = t - 1000$.

Pokud e_x, e_y a e_t značí rozsahy hledaného rekonstrukčního filtru r, v [6] je zaveden vztah mezi rozsahy a derivacemi v jednotlivých směrech, kde $e_x d_x = e_y d_y = e_t d_t$ a $V_s = e_x e_y e_t$. Samotné rozsahy jsou pak spočítány odděleně:

$$e_x = \sqrt[3]{\frac{V_s d_y d_t}{d_x^2}}, \quad e_y = \sqrt[3]{\frac{V_s d_x d_t}{d_y^2}} \quad \text{a} \quad e_t = \sqrt[3]{\frac{V_s d_x d_y}{d_t^2}}. \tag{3.18}$$

V [6] tvrdí, že rozsahy e_x, e_y a e_t by měly rovnoměrně pokrývat změnu barvy v každém směru, zjednodušeně, že rozsahy představují samotný objem V_s. To se sice jeví jako rozumný přístup avšak, jak je poznamenáno v [14], vyvstává závažný problém. Pokud nějaká vzorkovaná oblast obsahuje např. pouze jednu stejnou barvu, pak budou derivace d_x, d_y a d_t nulové a rozsahy — e_x, e_y a e_t — budou směřovat k nekonečnu. To však nejsou přijatelné výsledky. Proto je v [14] navržena alternativní rovnice, která upravuje rovnici (3.18) následovně:

$$e_x = \frac{V_s \cdot (1 - d_x)}{d_c}, \quad e_y = \frac{V_s \cdot (1 - d_y)}{d_c} \quad \text{a} \quad e_t = \frac{V_s \cdot (1 - d_t)}{d_c}, \tag{3.19}$$

kde $d_c = (1 - d_x) + (1 - d_y) + (1 - d_t)$. Ačkoliv je tento přístup přímočarý, dokáže se vypořádat právě s problémy, které způsobuje rovnice (3.18). Přesto existuje jeden případ, pro který i rovnice (3.19) selže. Pokud dochází k rapidní změně barvy z bílé na černou (nebo naopak), všechny derivace budou mít hodnotu 1 (tedy $d_x = d_y = d_t = 1$), potom ale bude $d_c = 0$ a opět budou všechny rozsahy směřovat k nekonečnu. Jak je poznamenáno v [14] jedná se o specifický případ, který je nutné řešit samostatně.

3.3.2 Výběr vhodného filtru

Cílem *adaptivní* metody je rekonstruovat obraz z různých vzorků. Tento přístup je podobný tzv. *rekonstrukci obrazu z neuniformních vzorků*[4]. Jak je ale uvedeno v [6], *bezsnímkové renderování* tento problém rozšiřuje o třetí časovou doménu, která činí celý proces komplikovanější. Proto je i v [6] přistoupeno k použití *Gaussova filtru* a obě domény jsou řešeny odděleně.

Aby to však bylo možné, je nejprve nutné rozdělit rekonstrukční filtr r — představený v úvodu této kapitoly — na dvě samostatné komponenty: prostorový filtr r_s a časový filtr r_t. Oba filtry je pak již možné definovat samostatně. A jak bylo řečeno, v [6] byl jako prostorový filtr použit dvourozměrný Gaussův filtr. Ten má přijatelné pásmové vlastnosti a je snadno implementovatelný. Navíc je možné využít faktu, že dvourozměrný filtr lze implementovat složením dvou jednorozměrných filtrů. Filtr r_s pak lze definovat jako:

$$r_s(\omega) = \frac{1}{2\pi\sigma_x\sigma_y} \cdot e^{-\frac{1}{2}\left(\left(\frac{\omega_x}{\sigma_x}\right)^2 + \left(\frac{\omega_y}{\sigma_y}\right)^2\right)}, \tag{3.20}$$

kde σ_x, resp. σ_y, značí standardní odchylku v x-ové, resp. y-ové, ose. Jak je patrné z rovnice (3.20) filtr r_s nerespektuje navržené rozsahy e_x a e_y. Je tedy potřeba upravit jeho rozměry. V [14] je pro tento účel využito tzv. pravidla 3σ, které říká, že 99,7% plochy pod Gaussovou křivkou spadá do intervalu mezi $(-3\sigma_{x/y}, 3\sigma_{x/y})$. Pak je možné nastavit $e_x = 3\sigma_x$ a $e_y = 3\sigma_y$ a rovnici (3.20) převést do tvaru:

[4] Více informací např. viz [12].

$$r_s(\omega) = \frac{9}{2\pi e_x e_y} \cdot e^{-\frac{9}{2}\left(\left(\frac{\omega_x}{e_x}\right)^2 + \left(\frac{\omega_y}{e_y}\right)^2\right)}. \tag{3.21}$$

Rekonstrukční filtr pro časovou doménu r_t je v [14] implementován obdobně, jako váhová funkce λ pro určení váhy *oblasti* obrazu U_i:

$$r_t(t_s) = e^{(t_s - t)/e_t s'}, \tag{3.22}$$

kde t je aktuální čas, t_s je čas poslední aktualizace pixelu (x, y) a s' je faktor, kterým je uměle upraven časový rozsah filtru. Ten je v [14] nastaven napevno na hodnotu 100. Je však nutné poznamenat, že by tato hodnota měla být variabilní pro různé typy scén. Přesto právě hodnota 100 dávala nejlepší výsledky pro většinu scén (viz [14]).

3.3.3 Rekonstrukční algoritmus

Pro rekonstrukci byl zvolen přístup zvaný *scatter*, implementovaný v [6]. Ten propaguje novou barvu vzorku jeho sousedům. Prostorový filtr r_s váží novou barvu vzorku podle jeho vzdálenosti od středu rekonstruovaných pixelů, které leží v nejbližším okolí vzorku daného rozměry filtru e_x a e_y. Váha barvy vzorku pak klesá s jeho vzdáleností od právě rekonstruovaného pixelu. Časový filtr r_t váží původní barvu rekonstruovaného pixelu podle jeho stáří a váha původní barvy klesá se zvyšujícím se věkem daného pixelu.

Aby byla rekonstrukce co nejefektivnější, byla využita rovnice z [14], která oba filtry spojuje dohromady a vážení je prováděno inkrementálně:

$$i(x_k, y_k, t) = \frac{1}{w}\left(\frac{9}{2\pi e_x e_y} e^{-\frac{9}{2}\left(\left(\frac{\omega_x}{e_x}\right)^2 + \left(\frac{\omega_y}{e_y}\right)^2\right)} \cdot i(x, y, t) + e^{(t_s - t)/e_t s'} \cdot w_k \cdot i(x_k, y_k, t_k)\right), \tag{3.23}$$

kde t_k označuje poslední čas aktualizace pixelu (x_k, y_k), w_k označuje poslední sumu vah obou filtrů a w je nová suma vah pro příští krok, která je vyčíslena podle rovnice:

$$w = \frac{9}{2\pi e_x e_y} e^{-\frac{9}{2}\left(\left(\frac{\omega_x}{e_x}\right)^2 + \left(\frac{\omega_y}{e_y}\right)^2\right)} + e^{(t_s - t)/e_t s'} \cdot w_k. \tag{3.24}$$

Kapitola 4

Implementace

Pro účely této práce byla souběžně vyvíjena aplikace, jejímž cílem bylo vhodným způsobem demonstrovat činnost *bezsnímkového renderování* v praxi na několika jednoduchých scénách. Kromě základní *bezsnímkové metody* byl kladen důraz i na demonstraci *adaptivní varianty*, a to jak řízeného vzorkování, tak následné rekonstrukce obrazu. Tato kapitola popisuje implementaci demonstrační aplikace a jejích jednotlivých částí.

Demonstrační aplikace byla vyvíjena v programovacím jazyce `C++` s důrazem na správné využití zásad OPP a výhod z nich plynoucích. Pro zobrazování výstupu *bezsnímkového rendereru* byl využita knihovna *OpenGL*[1]. Využití dalších knihoven a nástrojů bylo zvažováno, ale nakonec nepoužito.

4.1 Vytvoření a správa scény

Před vlastní implementací *bezsnímkového* rendereru bylo nutné zaměřit se na načtení a sestavení scény z jejího popisu. Jako vhodný formát pro popis scény byl vybrán formát `AFF`, jehož hlavní doménou je možnost definice animací.

4.1.1 Formát `AFF`

Pro výběr vhodného formátu pro popis scény bylo vzneseno několik hlavních požadavků, které musí daný formát splňovat. Kromě zobrazení základních primitiv, jako jsou trojúhelník, válec a koule, světel, materiálů a případně textur, byl hlavní důraz kladen na možnost definice animací ve scéně a to jak jednotlivých objektů tak i samotné kamery. Kromě toho byl hledán formát, který je co nejjednodušší, textový a není zatížen komerčními licencemi.

Všechna kritéria splňuje právě formát `AFF`, který byl nakonec zvolen a zakomponován. Formát `AFF`[2] vzniká v roce 2 000 rozšířením svého předchůdce, formátu `NFF` (z angl. *normal file format*), pro účely tzv. *BART* (z angl. *A Benchmark of Animated Ray Tracing*, viz [10]). Formát je textový a jeho struktura je velmi jednoduchá. Skládá se z tzv. entit, které umožňují definovat:

- kameru (pozice, směr pohledu, rozlišení obrazu v pixelech, apod.),
- světla (pozice, barva, možnost animací),
- materiály (difusní, ambientní a spekulární barva, index lomu, průhlednost, odlesky),

[1] Viz `http://www.opengl.org/`.

[2] Z angl. *animated file format*, definice a popis formátu na `http://www.ce.chalmers.se/BART/aff.html`.

- základní primitiva typu trojúhelník, válec a jehlan, koule a polygon.

Navíc je možné do sebe jednotlivé `AFF` soubory vnořovat pomocí tzv. *include* klauzule, čímž lze docílit opakování určité geometrie ve scéně, nebo lze definovat různé materiály a ty pak nastavovat jednotlivým objektům. Nejdůležitější entity ve formátu `AFF` jsou však transformace. Ty jsou přítomny dvě: statická a dynamická. *Statická* transformace umožňuje aplikovat transformační matici na všechny objekty (posun, rotace a změna měřítka), které obsahuje. Tato transformace je definována pomocí matice a její využití je např. ve spojení s *include* klauzulí, kde geometrie definovaná ve vkládaném souboru je posunuta na novou pozici ve scéně.

Druhá transformace je *dynamická* a je specifikována pouze unikátním jménem. Pomocí ní lze označit geometrii, která se bude animovat při spuštění scény. Samotný popis animace je pak přesunut do zvláštní entity, která je s dynamickou transformací svázaná jménem a animace je popsána pomocí transformací v klíčových snímcích v čase. Díky tomu lze dopočítat transformační matici pro konkrétní animaci podle aktuálního času běhu aplikace. Nutno poznamenat, že je možné oba typy transformací do sebe libovolně vnořovat a dynamických transformací lze definovat několik se stejným jménem.

4.1.2 Vlastní implementace a vytvoření *grafu scény*

Scéna je implementována ve třídě `Scene` a ve své podstatě se pouze jedná o chytrou obálku na data. Scéna v sobě zabaluje všechny potřebné seznamy (seznam všech primitiv, materiálů, textur, světel a animací), nastavení kamery, barvu pozadí a dále implementuje *graf scény*. Scéna potom nabízí *API*[3] jak pro její vytvoření (pomocí třídy `AffParser`, který se postará o rozparsování souboru s popisem scény), tak pro poskytování informací svému okolí (geometrie, kamera), nebo jejich úpravu (transformace geometrie při animacích).

Při jejím vytváření je konstruován tzv. *graf scény*, kde si každý uzel grafu (implementovaný třídou `SceneNode`) uchovává seznam všech svých potomků, seznam primitiv a svoji transformační matici. Celý graf pak udržuje informace o pozicích a transformacích jednotlivých primitiv v každém uzlu. Jednotlivé uzly grafu přímo odpovídají transformačním entitám z definice `AFF` souboru.

Zatímco u statického uzlu je transformační matice známa již při konstrukci scény (přímo z `AFF` souboru) a je možné ji uzlu nastavit ihned, dynamický uzel obsahuje pouze svoje jméno a jeho transformační matice je inicializována na matici identity[4]. Konkrétní transformační matice je vypočítána a nastavena až při vlastním běhu animace (viz podkapitola 4.1.3). Aby však transformace primitiv v konkrétním uzlu probíhaly správně, dochází k výpočtu finální transformační matice pro daný uzel skládáním matic v jednotlivých uzlech a to vždy směrem od listu ke kořenu (graf je implementován jako n-ární stom). Touto maticí jsou následně transformována všechna primitiva, která daný uzel obsahuje. Finální matici je nutné počítat vždy znovu pro každý statický (při sestavování scény) i dynamický (při každém snímku animace) uzel grafu.

4.1.3 Animace a jejich správa

Animace jsou popsány zvláštní entitou v `AFF` souboru, která obsahuje popis transformací v jednotlivých klíčových snímcích v čase. Při sestavování scény jsou vytvořeny příslušné

[3] API (z angl. *Application Programming Interface*) označuje rozhraní pro programování aplikací. Jedná se o sbírku procedur a funkcí, které lze využít mimo danou třídu nebo knihovnu.

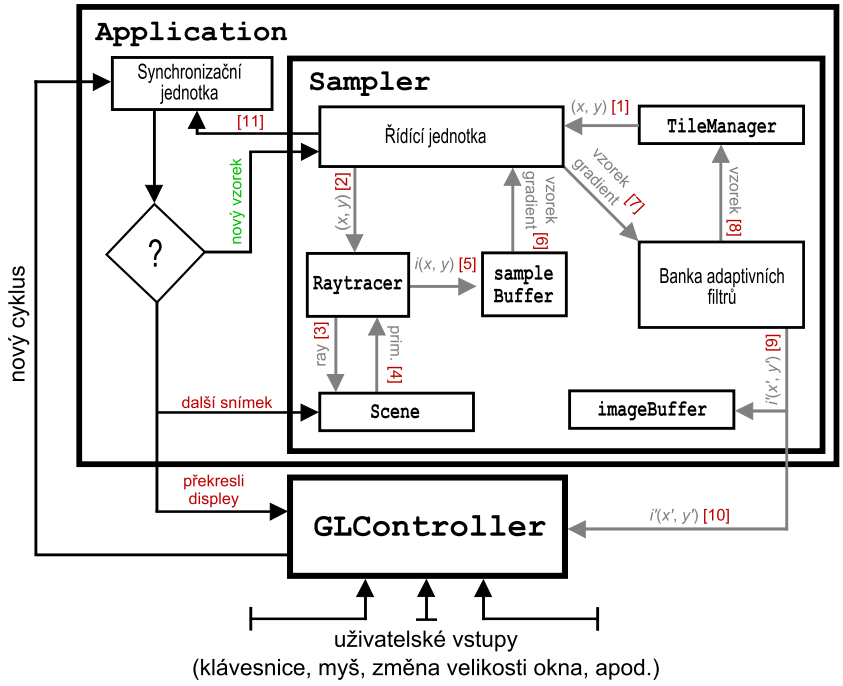
[4] Matice identity je jednotkový prvek pro násobení matic.

instance třídy `Animation`, které si v sobě uchovávají transformace v klíčových snímcích a na požádání jsou schopné poskytovat transformační matici v konkrétním časovém okamžiku. Pro její nalezení je provedena lineární interpolace mezi dvěma nejbližšími klíčovými snímky (podle času snímků a aktuálního času animace) a nová transformační matice je pak nastavena danému dynamickému uzlu. Nově získaná transformační matice je nejprve postupně skládána (násobena) se všemi maticemi rodičovských uzlů od aktuálního uzlu grafu. Transformace primitiv následuje vzápětí pomocí výsledné transformační matice.

Možných řešení problému animací bylo více. Např. nastavit objekt animace přímo dynamickému uzlu. Jelikož je však animace scény volitelná, byl zvolen přístup implementující tzv. *správce animací* (třída `AnimationManager`). Ten udržuje všechny animace ve scéně a na požádání vrací příslušné transformační matice podle jména dané animace a aktuálního času. Kromě toho si sám udržuje důležité proměnné pro řízení běhu animace celé scény (čas začátku a konce všech animací, počet snímků, aktuální čas animace a aktuální snímek). Dále pak poskytuje *API* pro řízení animace scény (přechod na následující nebo předchozí snímek, přechod na konkrétní snímek nebo na konkrétní čas animace).

4.2 Vlastní implementace aplikace

Implementace demonstrační aplikace vychází z obrázku 3.1, ale ve značné míře ho upravuje. Např. komponenta rekonstruktoru byla kompletně začleněna přímo do vzorkovače. Samotná aplikace se skládá ze tří hlavních komponent: `Application`, `GLController` a `Sampler` a její schéma je znázorněno na obrázku 4.1 včetně toku dat mezi jednotlivými bloky. Popis jednotlivých částí následuje v dalším textu této podkapitoly.



Obrázek 4.1: Blokové schéma demonstrační aplikace včetně toku dat mezi jednotlivými funkčními bloky a pořadím, v jakém jsou kroky vykonávány (v hranatých závorkách, červenou barvou). Schéma zobrazuje aplikaci pro *adaptivní* variantu *bezsnímkoveho renderování* a pracuje podle algoritmu 3.1.

4.2.1 Komponenty `Application` a `GLController`

Hlavní komponentou celé aplikace je komponenta `Application`, která je implementována ve stejnojmenné třídě. Třída `Application` řídí aktualizaci jednotlivých pixelů pomocí své *synchronizační jednotky*, která implementuje `while` cyklus z algoritmu 3.1 z řádku 3. Rozhodovací podmínka je znázorněna pomocí bloku s *otazníkem*, který podle aktuálního času běhu aplikačního cyklu a celkového času běhu aplikace posouvá animaci scény na následující klíčový snímek a pokud uběhl dostatečně dlouhý čas, nebo byl aktualizován požadovaný počet pixelů v daném kroku, vrací řízení zpět komponentě `GLController` (znázorněno červenou barvou). V opačném případě dochází k aktualizaci nového pixelu v rastru a řízení je předáno *vzorkovači* (zelená barva).

Implementaci rozhodovací podmínky pro synchronizaci s animací nebo ukončení aktuálního aplikačního cyklu bylo možné realizovat několika možnými způsoby:

1. Lze stanovit konstantní čas t_s, který je porovnán s dobou aplikačního cyklu aplikace t_r. Dokud bude platit nerovnost $t_r < t_s$, budou aktualizovány pixely pomocí vzorkovače. V okamžiku, kdy podmínka splněna nebude, dojde k přerušení cyklu a k předání řízení zpět komponentě `GLController`. Tímto způsobem je možné nastavit fixní obnovovací frekvenci, např. $25Hz$.

2. Čas t_s může být rovněž zvolen vhodně tak, aby odpovídal jednomu kroku animace uvnitř scény. Tím se *frame buffer* na displeji překreslí vždy, když je to nezbytně nutné, tedy při přechodu na nový snímek animace.

3. Další možností je nastavit fixní počet pixelů, které budou v jednom kroku aktualizovány. Doba běhu celého cyklu pak není řízena skutečným časem, ale pouze odezvou níže položených komponent (*vzorkovače* a *raytraceru*). Současně je však nutné synchronizovat čas animace a správně přecházet na následující snímek animace.

4. Posledním možným způsobem je obnovovat výstup na displeji vždy po přijetí uživatelských vstupů. Např. pohybem myši měnit pohled kamery ve scéně. Tento přístup je však velmi komplikovaný při spojení s dynamickými objekty a musí být kombinován s ostatními způsoby.

Komponenta `GLController` představuje vrstvu mezi aplikací a vykreslováním na displeji, které je realizováno pomocí *OpenGL*. `GLController` implementuje jednotlivé *callback* funkce pro obsluhu událostí spojených s uživatelskými vstupy (myš, klávesnice, změna velikosti okna, apod.) a rovněž řídí vykreslování aktuálního obsahu *frame bufferu*. To se děje v *callback* funkci `OnDisplay()`. V této funkci je nejprve předáno řízení aplikaci, která se postará o aktualizaci pixelů a synchronizaci s animací scény (viz výše). Jakmile je aktualizován požadovaný počet pixelů, nebo uběhl potřebný čas, je opět vráceno řízení zpět komponentě `GLController`, která se postará o vykreslení *frame bufferu*. S přihlédnutím k algoritmu 3.1, komponenta `GLController` implementuje hlavní nekonečnou smyčku na řádku 2.

Na první pohled by bylo možné obě výše uvedené komponenty spojit do jedné, avšak komponenta `GLController` byla vytvořena zejména proto, aby bylo možné oddělit logiku vlastního vykreslování od zbytku aplikace. Samo *OpenGL* je nyní využito pouze v této komponentě a zbytek aplikace je na něm nezávislý. Díky tomu je možné implementovat vlastní vykreslovací jednotku, která může být realizována zcela odlišně. Lze např. výstup vykreslovat pouze do obrázků a ukládat je na disk, nebo pro vykreslování využít jiných

grafických knihoven a *GUI* (např. *wxWidgets* nebo *Qt*), apod. A to implementováním jediné třídy bez větších zásahů do zdrojových kódy zbytku aplikace.

4.2.2 Vzorkovač (třída `Sampler`)

Srdcem *bezsnímkového* rendereru je vzorkovač implementovaný ve třídě `Sampler`. Ten v aktuální implementaci provádí vlastí vzorkování a aktualizaci pixelů a k tomu mu slouží dílčí komponenty potřebné pro vykonávání řízeného vzorkování a rekonstrukce obrazu. Řídící jednotka vzorkovače, které je předáno řízení z hlavní aplikace jako požadavek na aktualizaci nového pixelu, nejprve požádá o nové souřadnice třídu `TileManager` (ta realizuje vlastní řízené vzorkování a její detailní popis, viz podkapitola 4.3), která je získá a vrátí (krok 1 v obrázku 4.1 zobrazený červenou barvou).

Poté jsou v kroku 2 předány nově získané souřadnice *raytraceru*, který spočítá aktuální barvu daného pixelu vržením paprsku do scény (krok 3) a získáním průsečíku s nejbližším primitivem (krok 4). Nová barva, $i(x, y)$, je poté předána pomocnému *sample bufferu* (krok 5), který vytvoří nový vzorek na daných pozicích s aktuální barvou a aktuálním časovým razítkem jeho vzniku a ten si uchová. Současně pomocí nového vzorku jsou spočítány hodnoty gradientů v prostorové i časové doméně, které budou později sloužit pro výpočet rozměrů filtru při rekonstrukci (viz podkapitola 4.4). Nový vzorek a gradienty jsou vráceny zpět řídící jednotce (krok 6).

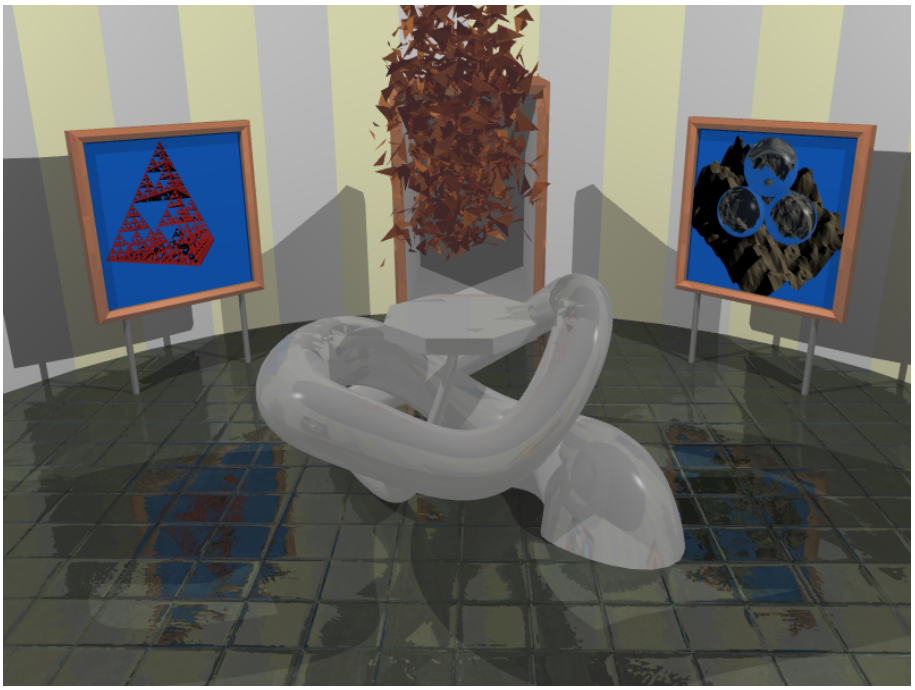
Nyní je možné provést vlastní rekonstrukci obrazu. Nový vzorek a právě spočítané gradienty jsou předány do *banky adaptivních filtrů* (krok 7), která reprezentuje rekonstruktor. V ní dojde k vypočítání rozměrů filtru a spolu s časovou složkou filtru je následně předán vzorek do `TileManageru` (krok 8) a je proveden přepočet pravděpodobností všech oblastí pro řízený výběr vzorku v příští iteraci. Současně dojde podle nově vytvořeného filtru k vážení sousedních pixelů okolo aktuálního vzorku přesně podle rovnice (3.23) a všechny okolní pixely s upravenými barvami $i'(x', y')$ jsou zapsány do vnitřního pomocného `imageBufferu` (krok 9), v němž je uchován aktuální výstup. Rovněž jsou změny promítnuty i do globálního *frame bufferu* uvnitř komponenty `GLController` (krok 10), která později provede jeho vykreslení (viz výše). Na závěr je řízení vráceno zpět ze vzorkovače hlavní aplikaci (krok 11).

4.2.3 Implementace jednoduchého *raytraceru*

Raytracer byl představen již v kapitole 2.6 a přesně podle tohoto teoretického základu byl i implementován. Jedná se pouze o jednoduchý *raytracer*, který byl oproštěn od všech složitých a časově náročných operací tak, aby zvládal základní požadovanou funkcionalitu (difuse, odlesky, odrazy a lomy, tvrdé stíny, textury a normálové mapy, řešení anti-aliasingu, apod.). Samotná implementace byla z části převzata z dřívějších projektů a z části inspirována v [16]. Větší část však spočívala v úpravě a přizpůsobení *raytraceru* pro potřeby této práce, zejména zakomponováním vlastní scény, která vychází z formátu `AFF` (viz výše).

V rámci vzorkovače *raytracer* úzce spolupracuje právě se scénou a na požádání vytvoří nový paprsek (krok 3) podle předaných souřadnic a hledá průsečík s nejbližším primitivem ve scéně podle aktuálního stavu scény, tedy podle aktuální geometrie. Hledání průsečíku je realizováno pomocí *kd*-stromu (viz podkapitola 2.6.2) nebo přímo procházením a testováním všech primitiv ve scéně. Současná implementace totiž nezohledňuje aktualizaci *kd*-stromu napříč animací, a proto je jeho využití omezeno pouze na scény se statickou geometrií (tato funkcionalita byla nad rámec této práce). Pokud je nalezeno nejbližší primitivum (krok 4), je pomocí rovnice (2.10) spočítána výsledná barva podle materiálu a osvětlení v bodě

Obrázek 4.2: První ukázka výstupu *raytraceru*, která ukazuje textury, *normálové mapy*, odlesky a průhledné koule na upravené scéně tzv. *Cornell boxu*. Scéna obsahuje celkem 32 primitiv. Se zapnutým *supersamplingem* 3×3 byla na rozlišení $1\,024 \times 1\,024$ renderována přibližně 10 minut.



Obrázek 4.3: Druhá ukázka výstupu *raytraceru* na scéně *Museum* z [10]. Scéna obsahuje animaci kamery a 1 024 animovaných trojúhelníků. Celkem scéna obsahuje 11 166 primitiv. Renderování scény na rozlišení 800×600 se *supersamplingem* 3×3 trvalo cca 40 minut.

průsečíku, případně jsou vytvořeny sekundární paprsky a ty jsou odraženy nebo zalomeny zpět do scény podle nastavení materiálu právě protnutého primitiva. Výsledkem je barva v aktuálním bodě rastru, která je vrácena zpět z *raytraceru* (krok 5).

Vlastní *raytracer*, společně se scénou a *kd*-stromem, byl implementován jako zcela samostatná jednotka, nezávislá na zbytku aplikace a lze ho tudíž použít i pro jiné účely nebo projekty. Přesto že je velmi jednoduchý, zvládá generovat kvalitní výstup v relativně krátkém čase (ukázka výstupu implementovaného *raytraceru*, viz obrázky 4.2 a 4.3).

4.3 Řízené vzorkování

O celé řízené vzorkování v rámci *adaptivní* varianty se stará komponenta `TileManager`. Jejím úkolem je nejprve rozdělit projekční rovinu na oblasti a na požádání vracet oblasti s nejvyšší pravděpodobností. A dále pak připočítat váhu nového vzorku a upravit rozložení pravděpodobností napříč oblastmi.

4.3.1 Dělení roviny a výběr vzorku

Při implementaci řízeného vzorkování bylo využito uniformní dělení obrazu na stejně veliké oblasti. O to se stará komponenta `TileManager`, která při své inicializaci vytvoří binární strom, kde každý uzel je reprezentován pomocnou strukturou `TTileNodeInfo`. Tato struktura si uchovává ukazatele na rodiče, oba pod-uzly a objekt `Tile` reprezentující konkrétní oblast U_i. Ta je přítomna pouze v listech stromu. Posledním atributem struktury je pravděpodobnost, jenž je součtem pravděpodobností obou pod-uzlů a nebo, pokud jde o list, je rovna pravděpodobnosti dané oblasti. Jelikož se jedná o binární strom, jsou pro jednoduchost povoleny pouze sudé počty oblastí horizontálně i vertikálně, přičemž počty se mohou lišit.

Samotný výběr konkrétní oblasti U_i pak spočíval v implementaci algoritmu τ z rovnice (3.5). Algoritmus pracuje nad výše zmíněným stromem a nechává propadnout náhodně generované číslo r od kořene až k listu, ve kterém je uložena konkrétní oblast U_i. Algoritmus τ je realizován metodou `GetTileWithinRange()` (viz zdrojový kód 4.1), která je implementována bez použití rekurze při průchodu stromem. Tato metoda vrací přímo oblast U_i (třída `Tile`) a teprve v ní jsou pak hledány souřadnice nového vzorku pomocí jedné z metod uvedených v kapitole 2.7. Tyto souřadnice jsou vráceny zpět vzorkovači. S přihlédnutím k diagramu na obrázku 4.1 se jedná o krok 1.

4.3.2 Aktualizace pravděpodobnosti

Finální rovnice pro výpočet heuristiky $h(U_i)$ a váhové funkce λ byla představena v kapitole 3.2.3, konkrétně se jedná o rovnici:

$$\sigma^2(U_{i,a+1}) = \overline{i^2}_{a+1} - \bar{i}_{a+1}.$$

Její hlavní výhodou je konstantní časová složitost $\mathcal{O}(1)$ a iterativní způsob, pro který je nutné pro každou oblast U_i uchovávat pouze poslední sumu vah (`ageWeightSum`) a poslední čas výběru vzorku z dané oblasti (`lastSampleTime`). Aktualizace pravděpodobnosti aktuálně vybrané oblasti je inicializována z *banky adaptivních filtrů*, která představuje zjednodušeně rekonstruktor a s přihlédnutím k diagramu 4.1 se jedná o krok 8. Komponenta `TileManager` nejprve přidá barvu nového vzorku do dříve vybrané oblasti v kroku 1 (metoda

```
// Metoda pro získání oblasti U_i pro výběr vzorku pomocí náhodného čísla r
Tile *TileManager::GetTileWithinRange()
{
    TTileNodeInfo *currNode = tilesTree[0];   // vybraný uzel stromu, kořen
    float r = random() * currNode->variance;  // náhodné číslo r

    // Hledá odpovídají oblast podle čísla r a jejich pravděpodobnosti
    for(int i = 0; i <= height; i++) {
        // Pokud se již jedná o list, vrací požadovanou oblast
        if(currNode->tile != NULL) return currNode->tile;

        // Zkouší levý podstrom
        if(r < currNode->left->variance) currNode = currNode->left;
        // Zkouší pravý podstrom
        else {
            r -= currNode->left->variance;
            currNode = currNode->right;
        }
    }

    // Pravděpodobně došlo k chybě jelikož nebyla nalezena žádná oblast
    return NULL;
}
```

Kód 4.1: Zdrojový kód metody `GetTileWithinRange()`, která implementuje algoritmus τ (rovnice (3.5)) a nerekurzivně prochází binárním stromem. Podle pravděpodobností jednotlivých uzlů hledá správnou oblast obrazu, kterou následně vrací. Hledání oblasti je realizováno pomocí náhodně generovaného čísla r v rozsahu 0 – max. pravděpodobnost kořenového uzlu.

`AddSample()` třídy `Tile`, viz zdrojový kód 4.2) a následně provede aktualizaci pravděpodobností všech uzlů stromu od daného listu s oblastí k jeho kořenu (metoda `UpdateVariances()` třídy `TileManager`).

Klíčový parametr metody `AddSample()` je parametr `scale`, který představuje faktor s váhové funkce λ z rovnice (3.10) pro určení váhy původní pravděpodobnosti oblasti v čase od jejího posledního výběru. Čím vyšší je faktor s, tím starší vzorky mají větší váhu a tím bude i vyšší pravděpodobnost pro výběr dané oblasti, naopak pro malé hodnoty faktoru s nemá původní pravděpodobnost žádný přínos a nová hodnota vzorku výslednou pravděpodobnost ovlivní daleko více. Pokud je $s = 1$ dochází prakticky k úplnému potlačení řízeného vzorkování. Graf na obrázku 4.4 ukazuje závislost váhy v čase podle faktoru s.

4.4 Rekonstrukce

V [6] je celá rekonstrukce oddělená od vlastní implementace a přesunuta na samostatnou výpočetní jednotku, na grafickou kartu. Tímto krokem je sice výrazně zvýšen výpočetní výkon, ale na druhou stranu rovněž porušen základní princip *bezsnímkového renderování*, které je nyní realizováno „snímkově“, neboť rekonstruktor musí čekat na získání barvy nového vzorku z raytraceru a teprve před zobrazením nového snímku na obrazovce je celá

```
// Přidá váhu vzorku do pravděpodobnosti dané oblasti
void Tile::AddSample(const TSample* const smpl, const float &scale)
{
    // Výpočet váhy pro vážení barvy vzorku
    float factor = smpl->color.Grayscale();  // barva v odstínech šedi
    float weight = expf((lastSampleTime - smpl->time) / scale) * ageWeightSum;

    // Spočítá očekávané hodnoty i' and i'^{2} pro daný krok
    expVal = ((weight * expVal) + factor) / (weight + 1.0);
    expValSqr = ((weight * expValSqr) + (factor * factor)) / (weight + 1.0);

    // Aktualizuje pravděpodobnost oblasti
    variance = expValSqr - (expVal * expVal);          // Steinerův teorém
    variance = (variance <= 0.01) ? 0.01 : variance;  // Ošetření vyhladovění

    // Aktualizuje sumu vah a časové razítko pro další iteraci
    ageWeightSum = (weight + 1.0);
    lastSampleTime = smpl->time;
}
```

Kód 4.2: Implementace rovnice (3.9) s využitím její inkrementální podoby z rovnice (3.15) pro aktualizaci pravděpodobnosti dané oblasti U_i, která je volána po každém spočítaném vzorku.

rekonstrukce provedena. Proto byla v demonstrační aplikaci rekonstrukce přímo začleněna do vzorkovače a je realizována komponentami `sampleBuffer` a *bankou adaptivních filtrů*.

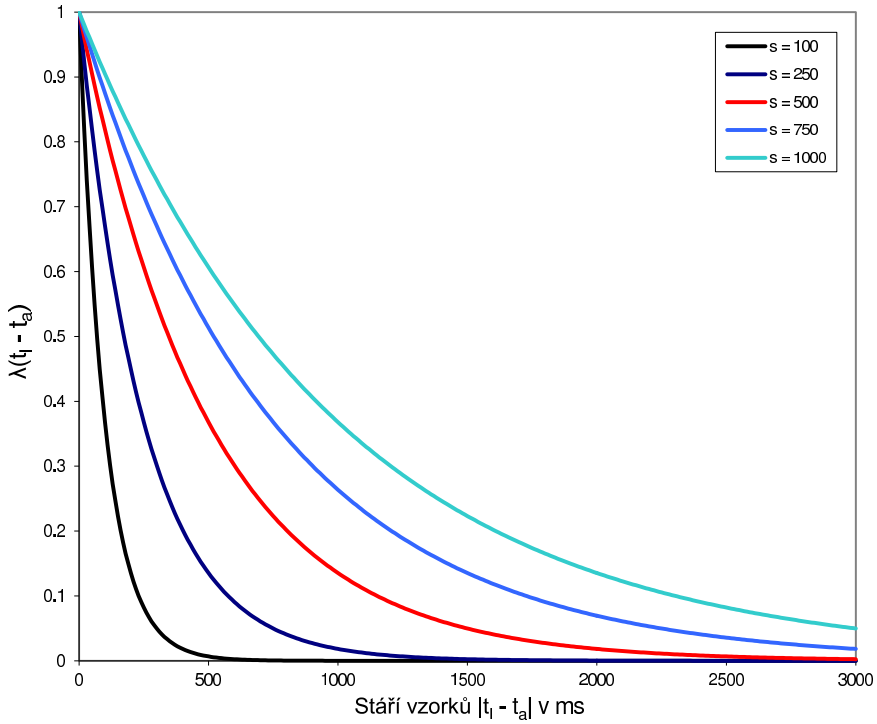
Přesto, že by se mohlo zdát, že implementace rekonstrukce obrazu bude spočívat pouze ve finální rovnici (3.23), není tomu tak a při její implementaci bylo nutné řešit několik dílčích problémů. Jednalo se o výpočet derivací, výpočet objemu V_s a řešení anti-aliasingu.

4.4.1 Aproximace směrových derivací, časová derivace

Výpočet směrových derivací d_x, d_y a d_t je realizován komponentou `sampleBuffer`, která uchovává všechny vzorky napříč rastrem. Samotný výpočet derivací však není triviální záležitost a v počítačové grafice nebývají derivace počítány přímo, ale aproximují se. V [6] aproximují prostorové derivace d_x a d_y výběrem čtyř nových vzorků, které leží na sousedních pozicích okolo aktuálního vzorku horizontálně a vertikálně, při rekonstrukci každého vzorku tedy vytvářejí kříž (tzv. *crosshair*) z 5 vzorků. Pro výpočet časové derivace berou další, šestý, vzorek na úplně nové pozici v čase. Na všechny vzorky jsou pak aplikovány diferenční filtry pro správnou aproximaci derivací. Tento přístup je jistě správný, avšak časově velmi náročný (5krát je volán *raytracer* pro výpočet barvy pomocných vzorků).

Naopak v [14] je postup velmi zjednodušený. Pro aproximaci derivací je využit *Sobelův operátor* na aktuální obsah `sampleBufferu` v odstínech šedi. Tento postup byl zvolen i v této práci. Prostorové derivace jsou počítány pomocí konvolučních masek:

$$s_x = \frac{1}{8}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad \text{a} \quad s_y = \frac{1}{8}\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}. \tag{4.1}$$



Obrázek 4.4: Váhová funkce $\lambda(t_l - t_a) = e^{(t_l - t_a)/s}$ z rovnice (3.10) pro různé hodnoty faktoru s. Z obrázku je patrné, že pro malé hodnoty s jsou starší vzorky méně významné pro celkovou pravděpodobnost oblasti než pro větší hodnoty faktoru s. Např. pro $s = 100$ dosahuje váha původní pravděpodobnosti po čtvrt vteřině méně než 10%. Na druhou stranu, hodnota $s = 1000$ znamená váhu pravděpodobnosti okolo 10% i pro oblasti, které nebyly vzorkovány tři vteřiny. Ideální se jeví prostřední červená křivka pro $s = 500$, jak se ale ukázalo při testování aplikace, záleží zejména na typu scény.

Tento přístup je nejen jednoduchý, ale rovněž velmi rychlý. Převod barev do odstínů šedi je prováděn pomocí rovnice (3.11). Výpočet prostorových derivací je realizován aplikováním příslušné masky na osmi-okolí pixelu, ze kterého byl vybrán aktuální vzorek. Časovou derivaci lze spočítat obdobně. Nejprve je nutné získat rozdíl barev v aktuálním a posledním platném čase, ideálně opět z osmi-okolí daného pixelu. A na tyto hodnoty pak aplikovat *binomiální filtr* b_t:

$$b_t = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \quad \longrightarrow \quad b_t = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}. \tag{4.2}$$

Pro výpočet časové derivace je nutné uchovávat dvě hodnoty, aktuální a minulou barvu v každém pixelu. Z tohoto důvodu obsahuje komponenta `sampleBuffer` dvě pomocná pole o rozměrech $M \times N$ pixelů a střídavě do nich zapisuje nové hodnoty vzorků, kde vždy jedno pole (nikoliv nutně první z nich) na pozici (x, y) obsahuje aktuální barvu a druhé pole obsahuje na stejné pozici poslední platnou barvu. Ta bude při další aktualizaci daného pixelu přepsána novou, budoucí platnou barvou.

Jak se ale ukázalo, náhodný výběr vzorků napříč rastrem, resp. napříč jedním pixelem

(pozice vzorku dosud nebyla nijak omezena a mohla ležet kdekoliv uvnitř pixelu) není pro výpočet derivací vhodný. Lepší a stabilnější variantou pro aproximaci derivací je zvolit pozici vzorku tak, aby ležela vždy v levém horním rohu každého pixelu. Jelikož je však jedním z požadavků na rekonstrukci obrazu provádění *anti-aliasingu* tam, kde je scéna delší dobu statická, znamená to počítat vždy vzorky dva: první na libovolné pozici uvnitř pixelu (právě pro řešení *anti-aliasingu*, viz podkapitola 4.4.3) a druhý vždy v levém horním rohu pixelu (pro výpočet prostorových a časových derivací). Z toho důvodu obsahuje komponenta `sampleBuffer` celkem dvě pomocná trojrozměrná pole. Obě mají rozměry $M \times N$. Jedno má hloubku nastavenu na 2 pro ukládání vzorků pouze v odstínech šedi na pozicích v levém horním rohu příslušného pixelu pro aproximaci derivací, tak jak bylo popsáno výše. Druhé pole, pro ukládání skutečných vzorků, které budou zapsány do výstupního image bufferu (pro řešení anti-aliasingu), má hloubku s_d.

4.4.2 Výpočet objemu V_s a vzorkovací frekvence $R(U_i)$

Objem rekonstrukčního filtru bude udávat velikosti jeho rozsahů a je počítán podle rovnice (3.17). Co tedy zbývá vypočítat je vzorkovací frekvence $R(U_i)$ pro každou oblast. Ta byla definována jako počet vzorků aktualizovaných v oblasti U_i v časovém intervalu $\langle t_h, t\rangle$, kde t_h označuje poslední platný čas. Vzorkovací frekvenci lze počítat podle rovnice:

$$R(U_i) = \frac{|S|}{m \cdot n}, \quad \text{kde} \quad S = \{i(x, y, t_k) | t_k > t_h\}, \tag{4.3}$$

kde m udává šířku oblasti U_i v pixelech a n pak její výšku. Lze si povšimnout, že v okamžiku, kdy platí $R(U_i) > 1$, dochází automaticky k supersamplingu obrazu a tedy k *anti-alisingu*.

Výpočet vzorkovací frekvence je realizován pomocí kódu 4.3, který využívá pomocného pole `recentSamplesTime`. V něm jsou za sebou uloženy časy všech vzorků vybraných v dané oblasti. Dále je použit ukazatel na index posledního platného vzorku `lastValidSampleIdx`. Počet všech platných vzorků je pak počet vzorků od ukazatele `lastValidSampleIdx` a koncem pole. Metoda `UpdateSamplingRate()` třídy `Tile` provede aktualizaci čítače platných vzorků vždy po přidání nového vzorku v kroku 8. Proměnná `now` obsahuje aktuální čas a proměnná `timeSpan` časový úsek, po který jsou vzorky platné, tedy z přihlédnutím ke kapitole 3.3.1 např. 1 000 milisekund.

```
// Aktualizuje počet platných vzorků v oblasti
void Tile::UpdateSamplingRate(const unsigned &timeSpan, const unsigned &now)
{
    unsigned i = lastValidSampleIdx;
    while(validSamplesNum > 0 && (now - recentSamplesTime[i]) > timeSpan) {
        validSamplesNum--;
        i++;
    }
    lastValidSampleIdx = i;
}
```

Kód 4.3: Aktualizace vzorkovací frekvence $R(U_i)$.

4.4.3 Řešení *anti-aliasingu*

Pro dosažení *anti-aliasingu* v *bezsnímkovém* rendereru není žádoucí počítat několik vzorků pro každý pixel, tak jako např. v *raytraceru*, ale získat ho zadarmo, tam, kde je scéna delší dobu statická. Jak se ukázalo, libovolná pozice vzorku uvnitř pixelu není vyhovující. Lepším řešením je zvolit několik fixních pozic uvnitř pixelu jako *offset* od jeho levého horního rohu (např. náhodně nebo pomocí *Haltonovy* posloupnosti). Pro výběr offsetu pro konkrétní pixel (x, y) je implementován pomocný buffer ukazatelů opět o rozměrech $M \times N$, kde každá hodnota udává index do pole offsetů. Při aktualizaci konkrétního pixelu je pak zvolen příslušný offset, který je přičten k pozici pixelu a ukazatel v pomocném bufferu je posunut na další offset. Při další aktualizaci téhož pixelu pak bude vybrán nový offset. A jelikož je počet vzorků na pixel s_d — tedy velikost pole s offsety — konečný, jsou ukazatele posouvány cyklicky od 0 do $s_d - 1$. Jedná se o druhé pomocné pole o hloubce s_d přítomné v komponentě `sampleBuffer`, zmiňované výše. Tímto principem je zajištěn správný výběr pozice vzorku v oblasti jednoho pixelu a je dosaženo lepších výsledků při potlačení aliasů v obraze tam, kde je scéna statická.

Při implementaci se ukázalo jako vhodnější řešení rozdělit rekonstrukci a výpočet *anti-aliasingu* do dvou částí. Tam, kde je scéna statická nedochází v čase k žádným změnám a časová derivace d_t se blíží k nule. V tomto případě je pak vhodné nastavit parametry časového filtru r_t manuálně na nějakou větší hodnotu. Neboť tam, kde je scéna delší dobu statická, mají i vzorky staré např. $1,5$ vteřiny stále vysokou váhu a tím větší částí přispějí svojí barvou k novému vzorku. Jeho barva se bude lišit výrazně pouze na hranách, čímž dojde k jejich vyhlazení. Celá idea je nastíněna v kódu 4.4.

```
// Pokud je časová derivace dostatečně velká, provede se klasická rekonstrukce
if(dT > 0.0001) {
    // vlastní rekonstrukce
}
// V opačném případě jsou váhy nastaveny ručně a provádí se anti-aliasing
else {
    DrawSample(x, y, smpl, 1500.0f, 1.0f);
    tile->AddSample(smpl, 500.0f);
    tileManager->UpdateVariances(tile);
}
```

Kód 4.4: Rozdělení rekonstrukčního algoritmu na dvě části. Pokud se současná časová derivace na pozici (x, y) blíží nule, je provedeno ruční nastavení vah a faktoru s a řešen *anti-aliasing*. V opačném případě je provedena rekonstrukce podle zdrojového kódu 4.6.

4.4.4 Rekonstrukční algoritmus

Samotný rekonstrukční algoritmus implementuje rovnici (3.23) a je reprezentován v *bankou adaptivních filtrů*. Ta z řídící jednotky vzorkovače v kroku 7 obdrží nový vzorek a gradienty, podle kterých vypočítá rozsahy nového filtru e_x, e_y a e_t. Nově vytvořený filtr nejprve přidá vzorek do dané oblasti pro aktualizaci rozložení pravděpodobností (krok 8). Následně iteruje nad všemi pixely v nejbližším okolí pixelu, ze kterého byl aktuální vzorek vybrán, a sečte jejich původní barvu váženou časovým filtrem r_t s barvou aktuálního vzorku, která je vážená prostorovým filtrem r_s. Nová barva na aktuální pozici v okolí pixelu $i'(x', y')$ je nejprve zapsána do pomocného *image bufferu* vzorkovače (krok 9) a rovněž je zapsána do

```
// Přidá novou barvu do frame bufferu, předtím na ni aplikuje váhy
void Sampler::DrawSample(int &x, int &y, TSample* smpl, float &scale, float &sw)
{
    // Původní barva pixelu a výpočet časové váhy 'tw'
    unsigned index = x+y*resolution.x();
    Color clr = imageBuffer->GetPixel(x, y);
    float tw = expf((lastTimes[index] - smpl->age) / scale) * ageWeights[index];

    // Váží novou barvu vzorku s původní barvou pixelu pro každou složku odděleně
    clr.red = ((sw*smpl->color.red) + (tw*clr.red)) / (sw + tw);
    clr.green = ((sw*smpl->color.green) + (tw*clr.green)) / (sw + tw);
    clr.blue = ((sw*smpl->color.blue) + (tw*clr.blue)) / (sw + tw);

    // Aktualizuje váhy a nastaví nový čas aktualizace pixelu
    ageWeights[index] = sw + tw;
    lastTimes[index] = smpl->age;

    // Nastaví novou barvu pixelu (x, y)
    imageBuffer->SetPixel(x, y, clr);
}
```

Kód 4.5: Metoda `DrawSample()` třídy `Sampler`, která se stará o vážení nové barvy vzorku prostorovou vahou (`sw`) a původní hodnoty pixelu časovou vahou (`tw`). Po přepočtu všech barevných složek je aktualizována suma vah (buffer `ageWeights`) a poslední čas aktualizace daného pixelu (buffer `lastTimes`). Jak je vidět, metoda `DrawSample()` je velmi podobná metodě `AddSample()` objektu `Tile` z kódu 4.2.

globálního *frame bufferu* v komponentě `GLController` (krok 10), která později provede jeho vykreslení na displej.

Díky inkrementální formě rovnice (3.23) není zapotřebí žádných složitých struktur pro ukládání informací o starých vzorcích apod. Je však nutné uchovávat starou sumu vah w_k (`ageWeights`) pro daný pixel a poslední čas t_k (`lastTimes`), kdy byl daný pixel aktualizován. Tyto hodnoty je nutné uchovávat pro všechny pixely v rastru. To bylo implementováno pomocí dvou pomocných dvourozměrných polí, která jsou přítomna přímo ve vzorkovači a *banka filtrů* s nimi pracuje. Implementace výpočtu nové barvy pixelu zapojením obou filtrů je znázorněna v kódu 4.5 v metodě `DrawSample()`. Výpočet nové barvy je nutné provést celkem třikrát pro všechny její složky. Celý průběh rekonstrukce, včetně získání derivací, výpočtu objemu filtru a jeho rozsahů a iterování nad příslušnými pixely je pak nastíněno v kódu 4.6.

4.5 Možná rozšíření a budoucí vývoj

Současný stav demonstrační aplikace je jistě dostačující, avšak ne zcela optimální. Již při návrhu bylo upuštěno od všech pokročilejších metod a implementace se zaměřila na hlavní aspekty *bezsnímkové* metody a její *adaptivní* varianty. Současná aplikace dokáže zobrazovat grafiku v reálném čase, avšak s poměrně omezeným množstvím pixelů aktualizovaných v jednom kroku. Oproti [6], kteří tvrdí, že dokáží generovat až 400 000 vzorků za sekundu, jich tato aplikace zvládá okolo 110 000 pro základní *bezsnímkovou* metodu a řízené vzorko-

vání. Rekonstrukce celý proces výrazně zpomaluje, především nutností počítat nový, druhý vzorek pro aproximaci derivací. V tomto případě je aplikace schopna zvládat aktualizovat 60 000 vzorků za vteřinu.

Dalším aspektem pro kvalitu zobrazení je velikost rastru, do kterého jsou pixely ukládány. Přijatelné výsledky jsou pro rozlišení 256×256, stejně jako tomu bylo i v [6]. Pro větší rozlišení by bylo nutné aktualizovat více pixelů a jelikož to není možné, obraz pak trpí větším výskytem artefaktů, případně je více rozmazán rekonstrukčním filtrem. Přesto je možné konstatovat, že aplikace splňuje všechna kritéria stanovená před započetím její implementace a na jednoduché scény je použitelná i v praxi.

4.5.1 Možná rozšíření a vylepšení

Jak již bylo řečeno, implementace se zaměřila pouze na základní aspekty metody *bezsnímkového renderování*. Proto je možné předpokládat další vývoj a vylepšení aplikace novými poznatky a přístupy. A jelikož je i sama metoda *bezsnímkového renderování* stále poněkud opomíjena, je možné očekávat bouřlivý vývoj i u ní. Několik uvažovaných možných vylepšení je představeno zde.

Neuniformní dělení obrazu: tento přístup byl implementován v [6] a byl diskutován v 3.2.1. Vlastní přínos pro řízené vzorkování by mohl spočívat v lepším a přesnějším výběru vzorků. Aktuální výstup v současné implementaci trpí především na rozhraní mezi dvěma sousedními oblastmi, kde jedna má velikou pravděpodobnost výběru (dochází v ní k nějaké animaci) a naopak druhá má pravděpodobnost malou. Pokud se animace přesune právě do druhé oblasti, je vidět „zub“ ve výstupu, neboť druhá oblast je stále vzorkována pomaleji a obsahuje velké množství neplatných vzorků. Pokud by však byly oblasti děleny podle prostorových derivací, tedy v okolí hran by bylo dělení jemnější, pravděpodobně by byly odstraněny znatelné přechody mezi jednotlivými oblastmi. Jak již bylo řečeno, problém tohoto přístupu může spočívat v neustálé aktualizaci rozdělení obrazu na oblasti a tím i ve výrazném zpomalení celé aplikace.

Rekonstrukční filtry: v současné implementaci byly zvoleny pouze ty nejzákladnější filtry pro rekonstrukci, přesněji *Gaussův filtr*. Stejně tak výpočet rozměrů a objemu filtru je značně zjednodušený. Ostřejší grafický výstup a lepší způsob nakládání se stářím vzorků by bylo jistě možné dosáhnout výběrem a implementací složitějších a sofistikovanějších filtrů. Přesto jsou zvolené filtry plně v souladu s dostupnými materiály a jsou implementovány stejně nebo podobně jako v [6] a [14].

Reprojekce vzorků: jedním z hlavních aspektů, které nebyly do implementace zařazeny je *reprojekce* a znovupoužití starých vzorků podle nové pozice kamery ve scéně. Tento přístup může v určitých typech scén výrazně zlepšit výstup rekonstruovaného obrazu (hlavně při pohybu kamerou) a urychlit výpočet samotný (zejména pokud je implementován na samostatné výpočetní jednotce). *Reprojekce* byla implementována v [6]. V této práci od ní bylo upuštěno zejména z důvodu zachování rychlé časové odezvy.

Rozdělení na více výpočetních jednotek: velmi zajímavý nápad, který by mohl výrazně snížit odezvu aplikace a metodu přiblížit k masovému využití. Jedním faktorem zde může být paralelizace samotného výpočtu (např. získání barvy vzorku z *raytraceru*). Dále pak může jít o rozdělení aplikace mezi různé výpočetní jednotky, jako tomu je u rekonstrukce v [6], která byla celá přesunuta a implementována pomocí *shaderů* na grafické kartě. Bohužel tento přístup přímo odporuje základní myšlence

bezsnímkového renderování a bez nových technologií v oblasti zobrazovacích zařízení ho nebude možné legitimně vyřešit.

Nové technologie v oblasti zobrazovacích zařízení: jak již bylo uvedeno v kapitole 2, současná zobrazovací zařízení nejsou schopna zobrazit výstup *bezsnímkového renderu* tak, jak byl navržen v [2] a je nutné tento fakt obcházet různými způsoby. Na vhodných zobrazovacích zařízeních by se jistě dalo využít plného potenciálu *bezsnímkové metody*. Jednou z možností jsou zařízení pro zobrazování virtuální reality nebo *HMD*. Další využití lze předpokládat u velikých displejů, kde je klasický přístup na vysokém rozlišení velmi neefektivní.

Aspekty založené na vnímání: tyto aspekty byly v rámci práce zcela ponechány stranou, přestože je největší potenciál této metody právě zde. Jak je uvedeno v [21] a v [22] veliký přínos metody leží právě v řízeném vzorkování, které upřednostňuje některé oblasti v obraze. Stejně se tomu děje i na sítnici v lidském oku a proto by bylo možné zohlednit výběr oblastí právě podle tohoto faktoru. Ve spojení s novými technologiemi zobrazovacích zařízení lze zcela logicky předpokládat, že právě zde leží budoucnost *bezsnímkové* metody, neboť jak se ukazuje, v rámci klasických displejů, kde zcela dominuje využití dvou bufferů, je její masové rozšíření a využití prakticky nereálné.

```
// Spočítá derivace v jednotlivých směrech
CalculateDerivatives(x, y, dX, dY, dT);

// Předpřipraví rozsahy filtru pro další výpočet
float ex = 1.0 - dX;
float ey = 1.0 - dY;
float et = 1.0 - dT;
float eSum = ex + ey + et;

// Spočítá objem filtru podle V_s = 1 / R(U_i) a upraví rozměry filtru
float Vs = pixelsPerTile / (float)(tile->GetValidSamplesNum() + 1);
ex *= (Vs / eSum); ey *= (Vs / eSum); et *= (Vs / eSum);

// Připraví konstanty pro vlastní rekonstrukci
float scaleFactor = 100.0 * et;
float gaussFactor = 9.0 / (2.0 * M_PI * ex * ey);
float exSqr = ex * ex;
float eySqr = ey * ey;

// Přidá nový vzorek do oblasti a aktualizuje pravděpodobnosti skrze celý strom
tile->AddSample(smpl, scaleFactor);
tileManager->UpdateVariances(tile);

// Pokryje všechny pixely v x-ové ose od -e_x do e_x
for(int dx = -(int)ex; dx <= (int)ex; dx++) {
    int cx = x + dx;
    if(cx < 0 || cx >= (int)resolution.x()) continue;
    float tx = smpl->pos.x() - dx - 0.5;
    tx = (tx*tx) / exSqr;

    // Pokryje všechny pixely v y-ové ose od -e_y do e_y
    for(int dy = -(int)ey; dy <= (int)ey; dy++) {
        int cy = y + dy;
        if(cy < 0 || cy >= (int)resolution.y()) continue;
        float ty = smpl->pos.y() - dy - 0.5;
        ty = (ty*ty) / eySqr;

        // Spočítá prostorou váhu filtru a vykreslí pixel
        float spatialWeight = gaussFactor * expf(-4.5 * (tx + ty));
        DrawSample(cx, cy, smpl, scaleFactor, spatialWeight);
    }
}
```

Kód 4.6: Ukázka algoritmu provádějícího vlastní rekonstrukci. Nejprve jsou vypočítány absolutní hodnoty derivací v jednotlivých směrech podle rovnic (4.1) a (4.2) a z nich jsou získány rozměry filtru podle rovnice (3.19). Následuje přidání vzorku do vybrané oblasti a přepočítání pravděpodobností všech oblastí. Poté jsou aktualizovány všechny pixely, které pokrývají rozměry filtru. Prostorová váha udává vzdálenost nového vzorku od středu každého filtrovaného pixelu. Samotné zapsání nové barvy pixelu a jeho vážení časovým filtrem je prováděno metodou `DrawSample()` zobrazené v kódu 4.5.

Kapitola 5

Testování a vyhodnocení

Testování demonstrační aplikace se zaměřilo především na kvalitu výstupu *bezsnímkového rendereru*, která byla měřena pomocí „ideálního“ rendereru – rendereru s nulovou odezvou. Ten byl schopen aktualizovat požadovaný počet vzorků v jednom časovém okamžiku simulací ubíhajícího času animace. Výkon aplikace je v tomto případě sporný, neboť již z podstaty metody *bezsnímkového renderování* je možné přímo nastavit požadovaný počet snímků za vteřinu, který je pak pevně daný po celý běh aplikace.

5.1 Testovací scény

Testovací scény byly zvoleny celkem tři (viz obrázek 5.1) a všechny jsou ve své podstatě velmi jednoduché, přesto rozličné a každá se zaměřuje na demonstraci jiného problému. První testovací scéna, *skákající míč*, je klasickým případem scény, který demonstruje pohyb předmětů ve statickém okolí. Scéna se skládá ze čtyř stěn, podlahy a stropu tvořících krychli. Pod stropem je umístěno jediné světlo. Uvnitř krychle je koule – míč –, který dopadá ze vzduchu na podlahu a odráží se zpět, přičemž se pohybuje zleva doprava. Scéna se skákajícím míčem má demonstrovat změnu geometrie scény v čase a to tak, že většina obrazu je po celou dobu animace statická a jen její minoritní část se mění. Důležitým faktorem v této scéně je stín vržený koulí na podlahu a stěny.

Druhá scéna, *planeta Země*, je ještě jednodušší a obsahuje jedinou kouli, která představuje planetu Zemi a jedno světlo (Slunce). Koule je potažena odpovídající texturou a v čase je statická. Dochází ale k pohybu kamery, která se otáčí zleva doprava a zpět, pak obraz oddálí a nakonec přiblíží. Cílem scény je zaznamenat pohyb všech předmětů ve scéně a jejich textur. Přesto že dochází pouze k pohybu kamery, v průběhu celé animace dojde k potřebě aktualizovat téměř všechny pixely rastru. Poslední scéna, *šachová figurka*, obsahuje jednu šachovou figurku z lesklého zlatého materiálu (2 400 trojúhelníků) a jedno světlo. Geometrie i kamera je statická, pohybuje se pouze světlo. Nejprve zleva doprava a následně zpět. Cílem scény je demonstrovat zachycení vrženého stínu v čase na těle figurky.

5.2 Metriky pro testování

Pro testování a vyhodnocení kvality jednotlivých částí bezsnímkové metody bylo nutné stanovit metriky, kterými lze určit výslednou chybu oproti ideálnímu obrazu. Při hledání metrik bylo vycházeno z [6], kde je porovnán výstup *bezsnímkového rendereru* v konkrétním časovém okamžiku s ideálním výstupem z *raytraceru*. Rozdíl intenzit jednotlivých pixelů

(a) *Skákající míč* (b) *Planeta Země* (c) *Šachová figurka*

Obrázek 5.1: Ukázka testovacích scén používaných v dalším textu pro demonstraci vizuálních výsledků jednotlivých částí *bezsnímkového rendereru*. Scény jsou zachyceny v konkrétním kroku jejich animace a renderovány na rozlišení 256×256 pouze pomocí *raytraceru* bez *supersamplingu*. Jedná se tedy o referenční „ideální“ výstupy.

pak udává výslednou chybu. Bezsnímkový výstup byl porovnáván vždy při přechodu na nový klíčový snímek animace. Tento postup je velmi jednoduchý a intuitivní, ale rozhodně není špatný, neboť jak je uvedeno v [20], je v každém časovém okamžiku přítomný aktuální výstup *bezsnímkového rendereru*[1]. Samotné vyčíslení chyby bylo provedeno přes všechny snímky animace, kde pro každý snímek byla chyba počítána podle rovnice:

$$\sigma = \frac{1}{w \cdot h} \sum_{x=1}^{w} \sum_{y=1}^{y} i(x, y), \tag{5.1}$$

kde w udává šířku a h výšku obrazu v pixelech a $i(x, y)$ je intenzita barvy pixelu na dané pozici po rozdílu mezi oběma výstupy. Chyba tedy udává průměrný rozdíl v intenzitách barev mezi aktuálním výstupem *bezsnímkového rendereru* a ideálním obrazem. Jak již bylo řečeno, simulace probíhala pomocí rendereru s nulovou odezvou, výchozí metoda pro výběr vzorků pro řízené vzorkování a rekonstrukci byla *Haltonova* posloupnost a počet oblastí byl 32×32 a pro rekonstrukci byl zvolen faktor $s = 100$.

5.3 Základní varianta *bezsnímkové* metody

Testování základní varianty se zaměřilo na porovnání výstupu mezi jednotlivými metodami pro výběr vzorků pro aktualizaci, které byly představeny v kapitole 2.7. Vizuální výsledky jsou znázorněny na obrázku 5.3 a jednotlivé chyby pak v grafu 5.2a. Porovnáním klasických metod pro výběr vzorků lze dojít k závěru, že si jsou velmi podobné a některé dosahují dokonce horších výsledků než původně navrhovaný náhodný výběr vzorků. Ten dosahuje srovnatelných výsledků především díky kvalitnímu generátoru pseudonáhodných čísel, který byl implementován podle [16]. *Haltonova* posloupnost se skutečně ukazuje jako nejlepší napříč všemi testy a vizuální výsledky jsou rovněž nejlepší (stejně jako v [14]). Pokrytí projekční roviny je daleko lepší než u ostatních metod a nedochází ke shlukování vzorků.

[1] Ostatně s přihlédnutím k algoritmu 2.3 je vždy přítomen *frame buffer* a ten obsahuje celý, i když z větší části zastaralý snímek.

5.4 Řízené vzorkování

Pro testování řízeného vzorkování byla zvolena pouze *Haltonova* posloupnost jako vítězná metoda pro výběr nového vzorku k aktualizaci. Testování pak bylo rozděleno podle počtu oblastí v obou směrech a testovány byly počty 8×8, 32×32 a 64×64. Pro každý z těchto počtů pak byly testy dále děleny podle hodnoty faktoru s, která byla volena z množiny $\{100, 250, 500, 750, 1\,000\}$, kde hodnota $1\,000$ dává váhu zhruba 10% těm oblastem, jež nebyly vybrány tři vteřiny.

Vizuální výsledky jsou zobrazeny na obrázcích 5.4 a 5.5, z důvodu šetření místem jsou zobrazeny pouze ty nejzajímavější varianty. Přehled výsledků odchylky od ideálního výstupu je pak v grafu 5.2b. Dosažené výsledky jsou zde sporné. Pro první dvě testovací scény jasně vítězí počet oblastí 32×32 a hodnota faktoru $s = 250$. Pro třetí scénu, *šachová figurka*, však byly nejlepší výsledky naměřeny pro počet oblastí 8×8, což však vizuální výsledky nepotvrzují (viz obrázky 5.5e versus 5.5g). Faktor s pak dává pro všechny uvažované počty oblastí nejlepší výsledky pro hodnotu $1\,000$ což je způsobeno samotnou podstatou scény. Jelikož se mění pouze osvětlení a stín na těle figurky, i velmi staré vzorky mají stále platnou hodnotu a proto se pravděpodobnost pro výběr oblasti mění jen velmi zvolna. Tomu se děje zejména na okrajích figurky.

5.5 Rekonstrukce

Cílem rekonstrukce je odstranit nebo alespoň potlačit výskyt prostorového rozptylu rozmazáním obrazu. Pro testování bylo zvoleno nastavení různého počtu vzorků, které se budou aktualizovat v jednom kroku. Je jasné, že vyšší počty by měly dávat lepší výsledky a lépe se vypořádat s aliasy. A výsledky tomu jednoznačně odpovídají. Vizuální výsledky jsou znázorněny na obrázku 5.6 a velikost chyby je shrnuta v grafu 5.2c. Pro testování byl počet oblastí nastaven na 32×32, faktor $s = 100$ a počty vzorků byly voleny z množiny $\{2\,250, 4\,500, 9\,000, 18\,000\}$.

Nejlepších výsledků bylo podle očekávání dosaženo pro všechny testovací scény pro $18\,000$ vzorků na jeden iterační cyklus. Přesto je zajímavé sledovat rozdíly v chybě. Zatímco pro scénu *skákající míč* dává $9\,000$ vzorků daleko lepší výsledky oproti $4\,500$, rozdíl od $18\,000$ již není tak razantní a jelikož se jedná o dvojnásobnou zátěž, je nutné se zamyslet, zda výsledná kvalita odpovídá nepříjemnému snížení odezvy. Jinými slovy je nutné hledat ideální poměr mezi kvalitou a rychlostí výpočtu, ostatně jako u většiny aplikací zobrazujících počítačovou grafiku v reálném čase. Pro nižší počty jsou pak vizuální výsledky velmi nepřesvědčivé a výstup trpí daleko více artefakty nebo je více rozmazaný.

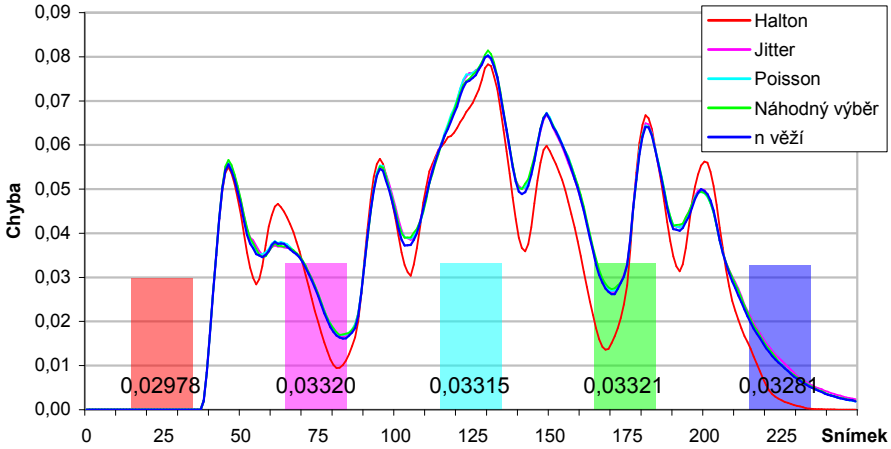
5.6 Zhodnocení výsledků

Jednotlivé vizuální výstupy pro všechny části *bezsnímkové* metody již byly představeny a diskutovány v předchozích částech této kapitoly, stejně jako míra chyby pro všechna možná nastavení a všechny testovací scény. Sama aplikace splňuje kritéria stanovená před začátkem jejího vývoje a přesto, že nedosahuje úplně 100% výstupu a není úplně nejefektivnější, zvládá aktualizovat okolo $60\,000$ vzorků v rámci rekonstrukce a zhruba dvojnásobek bez ní. Proto je možné její reálné využití pro zobrazování jednoduchých scén na nižším rozlišení.

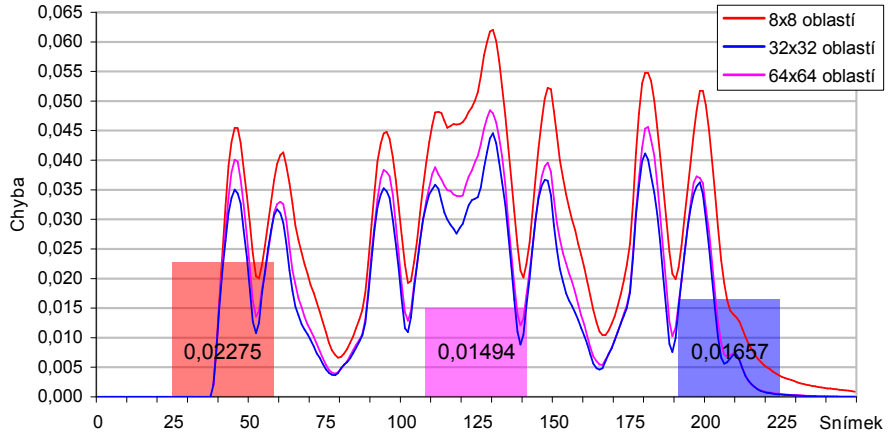
Hlavním cílem všech provedených testů bylo demonstrovat přínos *adaptivní* varianty do základní *bezsnímkové* metody. Sama *bezsnímková* metoda byla podrobena zkoumání

vlivu různých přístupů pro výběr nového pixelu vhodného pro aktualizaci. Jak se ukázalo, čistě náhodný výběr nestačí a ostatní navrhované metody dosáhly téměř shodných výsledků a stejně tak propadly. Naopak se jako zásadní krok ukázalo využití *Haltonovy* posloupnosti, která ve všech testovacích scénách dosahovala výrazně nižší chyby a vizuální výsledky byly kompaktnější a netrpěly shlukováním vzorků.

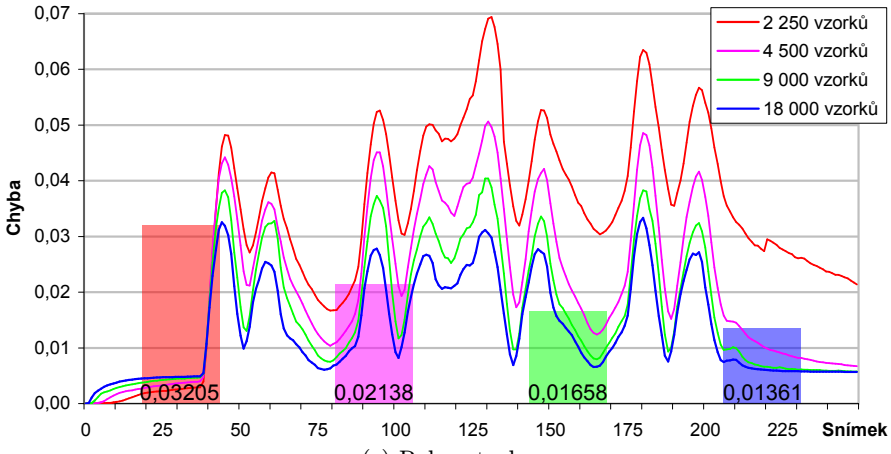
Řízené vzorkování pak s využitím *Haltonovy* posloupnosti přineslo další zlepšení výstupu, především odstraněním artefaktů a neplatných pixelů v oblastech rastru, kde je již scéna statická. Díky upřednostnění určitých oblastí rastru mohlo být vzorkování soustředěno právě tam, kde se odehrává nějaká animace, nebo scéna prochází razantní změnou geometrie nebo osvětlení. Cílem testů bylo demonstrovat rozdílné chování řízeného vzorkování v závislosti na zvoleném počtu oblastí v obou směrech a na hodnotě faktoru s. Precizním nastavením obou hodnot lze dosáhnout velmi kvalitních výsledků. Samotná rekonstrukce přináší další zkvalitnění výstupu (včetně řešení anti-aliasingu zcela zadarmo) a snížení celkové chyby. Kvalita výstupu pak závisí na celkovém počtu vzorků, které je aplikace schopna aktualizovat v jednom cyklu. Jak bylo ukázáno, pro 9 000 vzorků bylo dosaženo již přijatelných výsledků, přičemž se jedná o pouhých 14% z celkového počtu pixelů. Zde však bohužel leží kámen úrazu, neboť sama aplikace při reálném běhu dokáže aktualizovat pouze 2 500 vzorků v jednom průchodu, což výslednou kvalitu výstupu nepříjemně degraduje. Přesto je nutné znovu zopakovat, že metoda *bezsnímkového renderování* není metodou pro fotorealistické zobrazování počítačové grafiky.



(a) Základní metoda

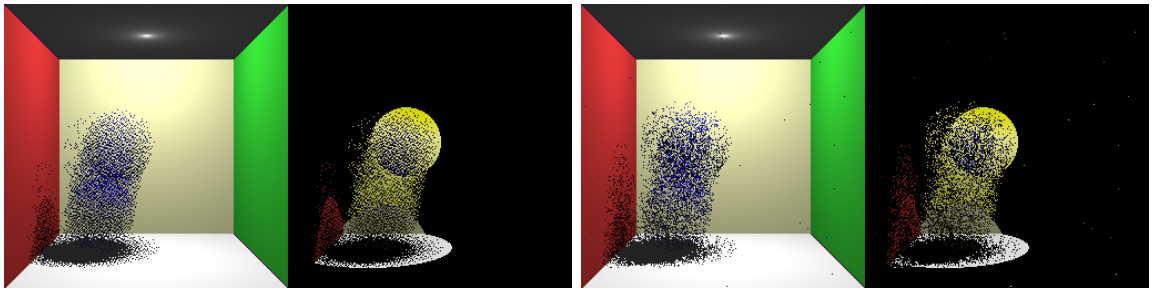


(b) Řízené vzorkování

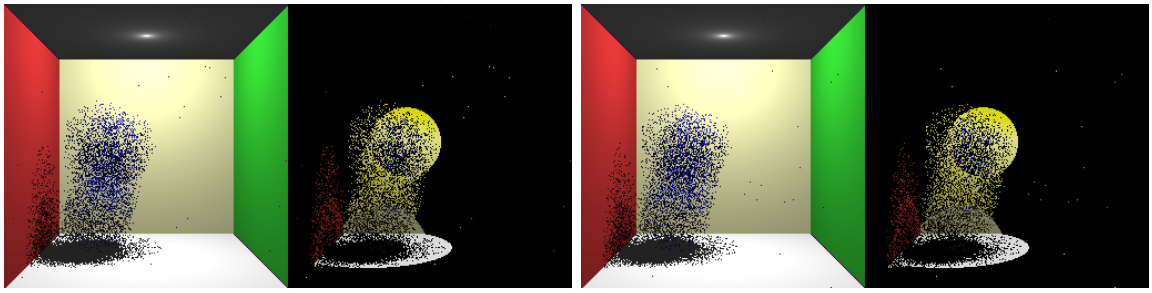


(c) Rekonstrukce

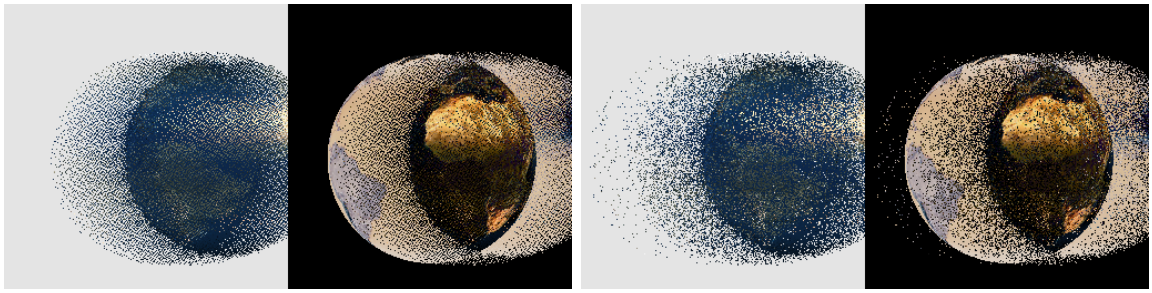
Obrázek 5.2: Grafy zachycující chybu v každém klíčovém snímku celé animace (testovací scéna *skákající míč*), ve sloupcích jsou průměrné hodnoty chyby pro celou animaci. Animace začíná na 38 snímku a končí na snímku 213. Pro základní metodu dosahuje nejlepších výsledků *Haltonova* posloupnost jejíž chyba je téměř po celou dobu animace nižší než u ostatních metod; ty dosahují prakticky shodných výsledků. Pro řízené vzorkování jsou rozdíly mezi různými hodnotami faktoru s minimální, v grafu je zobrazena chyba pro různé počty oblastí. U rekonstrukce je potvrzeno, že čím více vzorků je aktualizováno v jednom cyklu, tím je chyba nižší. Pro zbývající dvě scény jsou průběhy chyby podobné.



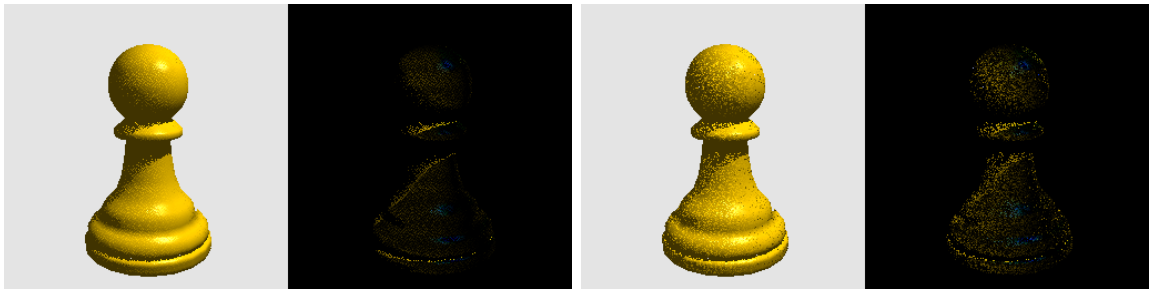
(a) *Haltonova* posloupnost (b) Náhodný výběr



(c) *Poissonovy disky* (d) *Jittered* vzorkování

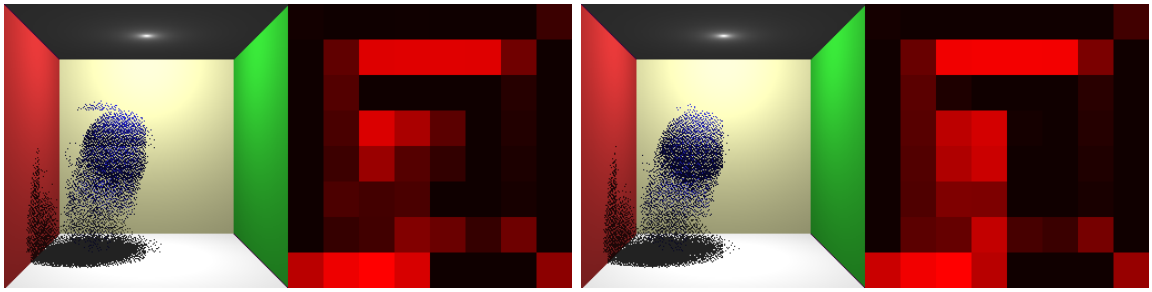


(e) *Haltonova* posloupnost (f) Náhodný výběr

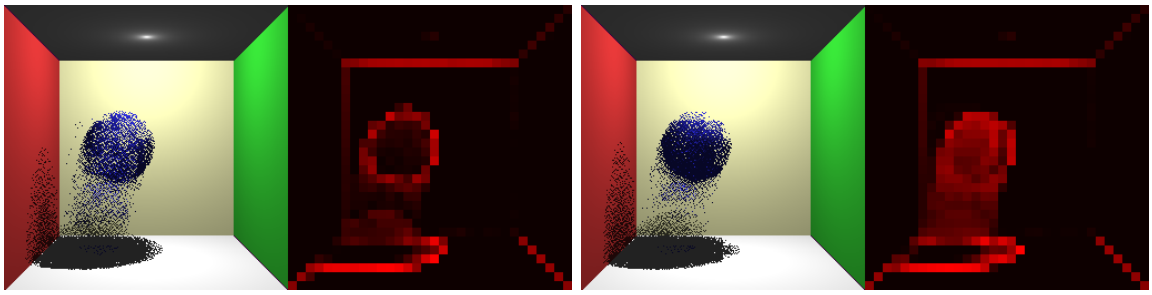


(g) *Haltonova* posloupnost (h) Náhodný výběr

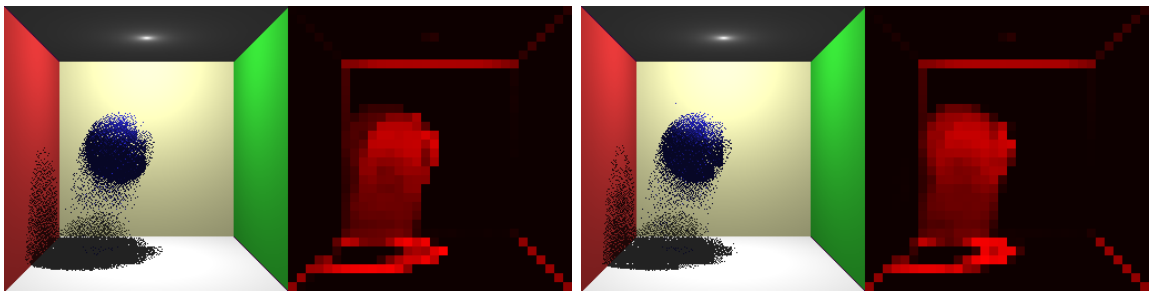
Obrázek 5.3: Vizuální výsledky základní *bezsnímkové* metody pro všechny testovací scény. V levém obrázku je zobrazen aktuální výstup rendereru, na obrázku vpravo potom jeho rozdíl s ideálním výstupem, získaným z *raytraceru*. Z obrázků je jasně patrný rozdíl mezi *Haltonovou* posloupností a náhodným výběrem pozice vzorku. Ostatní metody pro výběr pozice vzorku mají srovnatelné výsledky s náhodným výběrem.



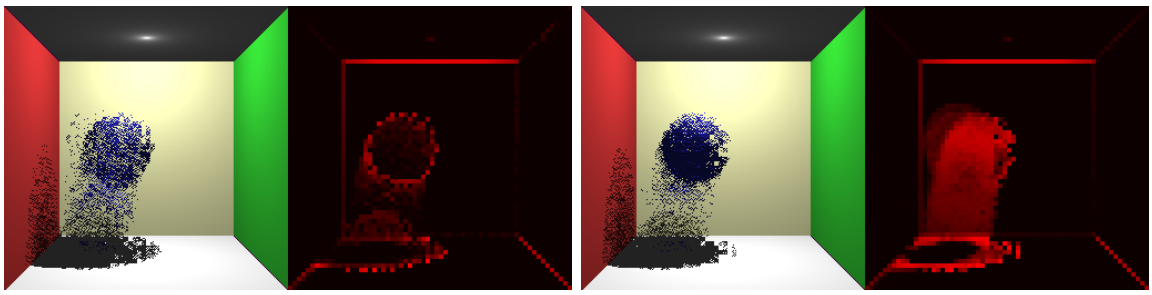
(a) $8 \times 8, s = 100$ (b) $8 \times 8, s = 500$



(c) $32 \times 32, s = 100$ (d) $32 \times 32, s = 250$

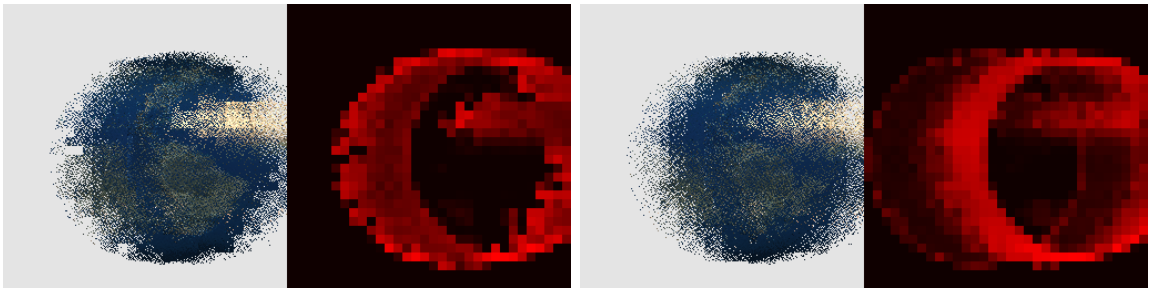


(e) $32 \times 32, s = 500$ (f) $32 \times 32, s = 1000$

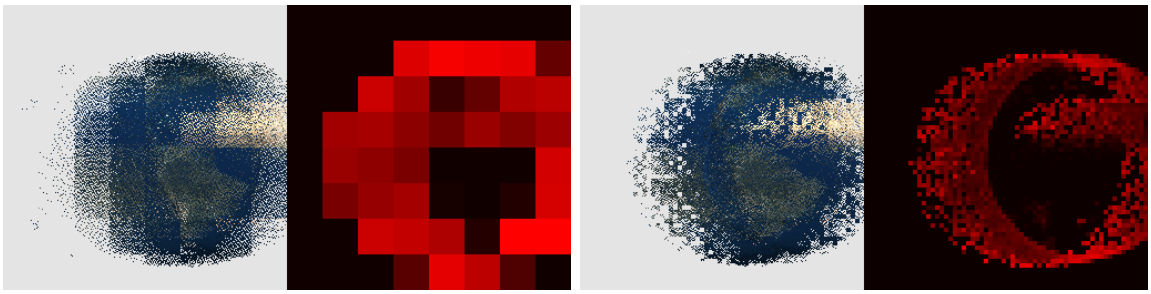


(g) $64 \times 64, s = 100$ (h) $64 \times 64, s = 500$

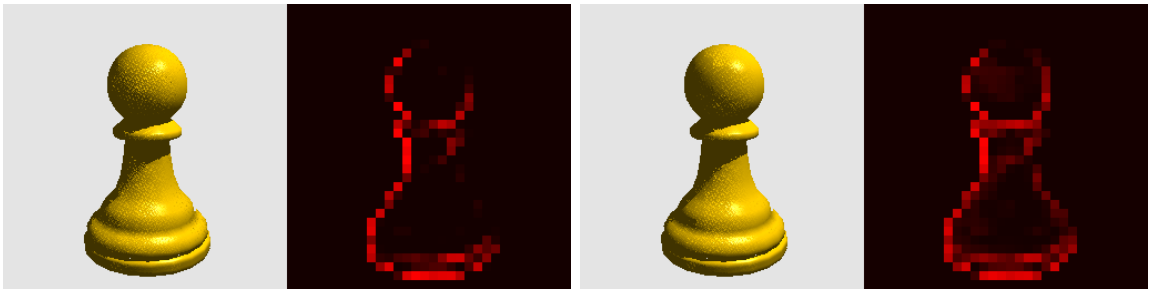
Obrázek 5.4: Řízené vzorkování pro scénu *skákající míč*. Obrázky ukazují rozdílné výsledky jak pro různý počet oblastí, tak pro různé hodnoty faktoru s (viz obrázek 4.4). Na každé dvojici obrázků je vlevo zobrazen aktuální výstup a vpravo pak rozložení pravděpodobnosti pro výběr dané oblasti v dalším kroku (zobrazeno červenou barvou pro vyniknutí rozdílů), kde maximální pravděpodobnost má hodnotu $1,0$. Nejlepších výsledků je dosaženo při rozlišení 32×32 pro faktor $s = 250$.



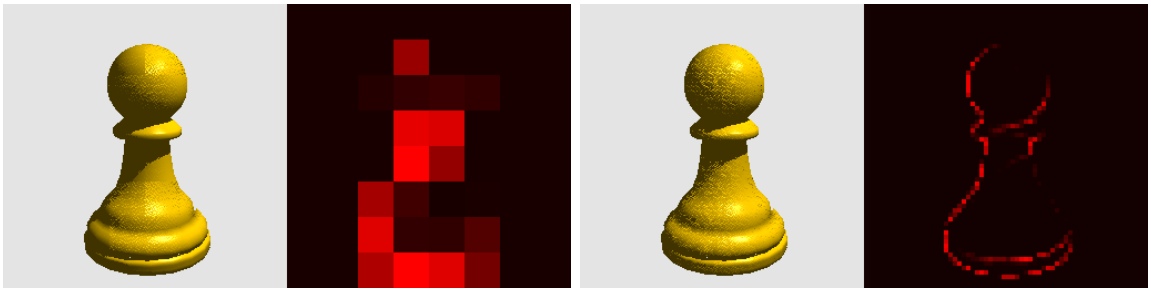
(a) $32 \times 32, s = 250$ (b) $32 \times 32, s = 1000$



(c) $8 \times 8, s = 250$ (d) $64 \times 64, s = 250$

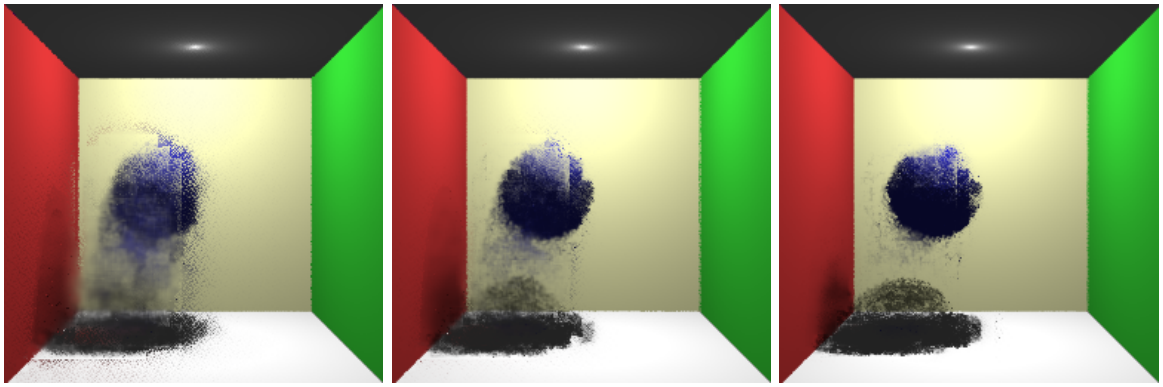


(e) $32 \times 32, s = 250$ (f) $32 \times 32, s = 1000$

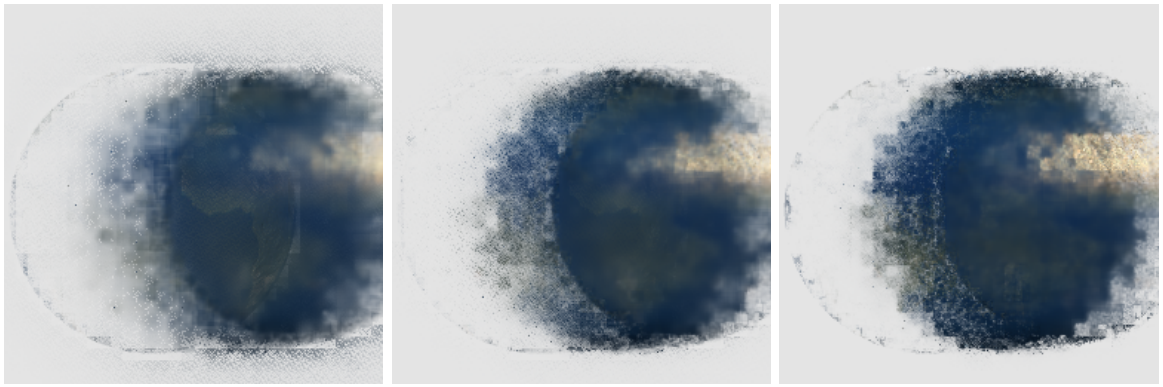


(g) $8 \times 8, s = 250$ (h) $64 \times 64, s = 250$

Obrázek 5.5: Řízené vzorkování pro zbývající dvě testovací scény. Obrázky opět ilustrují rozdílnou kvalitu pro různé počty oblastí a pro různé hodnoty faktoru s. Nejlepší výsledky jsou opět pro počet oblastí 32×32 pro obě scény. Faktor s je však vhodné volit podle typu scény. Zatímco u *planety Země* je opět ideální faktor $s = 250$ (změna geometrie je velmi rychlá), u scény *šachová figurka* je dosaženo daleko lepších výsledků pro faktor $s = 1\,000$, neboť nedochází k žádné změně geometrie, mění se pouze stín.



(a) 2 250 vzorků (b) 4 500 vzorků (c) 9 000 vzorků



(d) 2 250 vzorků (e) 4 500 vzorků (f) 9 000 vzorků

(g) 2 250 vzorků (h) 4 500 vzorků (i) 9 000 vzorků

Obrázek 5.6: Rekonstrukce obrazu pro všechny scény pro různé počty vzorků aktualizovaných v jednom kroku. Pro nižší počty vzorků je vizuální výstup rozmazaný a neaktuální a dokonce obsahuje větší množství šumu (dobře viditelné na obrázku 5.6g na pozadí okolo figurky). Pro 9 000 vzorků jsou výsledky relativně dobré a artefakty jsou při animaci jen stěží postřehnutelné. Např. u *šachové figurky* jsou rozeznatelné odlesky světla na povrchy figurky a okraje stínů jsou ostré, naopak pro 4 500 vzorků nejsou odlesky prakticky vidět a okraj stínu je velmi rozmazaný.

Kapitola 6

Závěr

Cílem této práce bylo nastudovat metodu *bezsnímkového renderování* včetně její *adaptivní* varianty a proniknout hlouběji do principů této metody. Přesto, že tato metoda byla navržena před více než 17 lety (viz [2]), jejímu vývoji a většímu zkoumání se prozatím věnovalo velmi málo pozornosti. Její studium bylo proto náročné a prakticky jsem mohl vycházet pouze ze tří publikací. V největší míře z [20], [6] a [14].

Praktická část práce se zaměřila na implementaci vlastní aplikace, která vhodným způsobem činnost této metody demonstruje. Pro aplikaci byl implementován jednoduchý *raytracer*, který se stará o výpočet nové barvy aktualizovaných pixelů a ty poskytuje *vzorkovači*. Raytracer byl navržen jako zcela nezávislá jednotka na zbytku aplikace a lze ho využít i samostatně u jiných projektů. Hlavním jádrem aplikace je pak *vzorkovač*, který se stará o vlastní *bezsnímkové renderování*, tedy o výběr pixelů vhodných k aktualizaci, tj. o řízené vzorkování, a následnou rekonstrukci obrazu.

Při návrhu aplikace byla značná část úsilí věnována především její efektivitě a kvalitě výstupu. Přesto nebyly implementovány některé pokročilé metody pro rekonstrukci obrazu, které jsou nastíněny v kapitole 4.5.1. Jelikož byla snaha implementovat „pravý“ *bezsnímkový renderer*, veškerý kód byl ponechán v rámci jedné výpočetní jednotky, fyzického procesoru počítače. Tím samozřejmě byla snížena celková odezva hlavně v rámci rekonstrukce.

Kapitola 2 se zaměřila na srovnání metody *bezsnímkového renderování* s tradiční metodou pro zobrazování 3D počítačové grafiky v reálném čase, která využívá přepínání mezi dvěma buffery. Dále se pak zabývala definováním metody samotné a faktorů podílejících se na kvalitě jejího výstupu, tak jak byla formálně popsána v [20]. Kapitola 3 se poté věnuje především *adaptivní* variantě, která byla představena v [6]. V kapitole jsou popsány všechny aspekty, kterými lze ovlivnit, resp. vylepšit, výstup *bezsnímkového rendereru* — jako je řízené vzorkování s výběrem vhodné oblasti obrazu a rekonstrukce obrazu. Kapitola rovněž představuje matematické rovnice popisující problematiku spojenou s adaptivní variantou na teoretické úrovni.

Kapitola 4 se věnuje vlastní implementaci demonstrační aplikace, jež byla praktickou součástí tohoto projektu v jeho letní části. Tato kapitola popisuje implementaci všech jednotlivých součástí výsledné aplikace, jako je popis a sestavení scény, implementace jednoduchého *raytraceru*, a pak samotnou *bezsnímkovou* metodu a všechny problémy, které je nutné při implementaci řešit. Na závěr diskutuje možný budoucí vývoj a vylepšení. Kapitola 5 pak diskutuje vizuální výsledky pořízené pří testování a vyhodnocuje provedené testy na připravených testovacích scénách.

Literatura

[1] Bergman, L.; Fuchs, H.; Grant, E.; aj.: Image rendering by adaptive refinement. *SIGGRAPH Comput. Graph.*, ročník 20, č. 4, Srpen 1986: s. 29–37, ISSN 0097-8930.

[2] Bishop, G.; Fuchs, H.; McMillan, L.; aj.: Frameless rendering: double buffering considered harmful. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, 1994, ISBN 0-89791-667-0, s. 175–176.

[3] Burr, D.; Ross, J.: Visual processing of motion. *Trends in Neurosciences*, ročník 9, č. 7, 1986: s. 304–307, ISSN 0166-2236.

[4] Dayal, A.; Watson, B.; Luebke, D.: Improving frameless rendering by focusing on change. In *ACM SIGGRAPH 2002 conference abstracts and applications*, SIGGRAPH '02, 2002, ISBN 1-58113-525-4, s. 201–201.

[5] Dayal, A.; Watson, B.; Luebke, D.; aj.: Temporally Adaptive Frameless Rendering. Technická zpráva, Northwestern University, 2004.

[6] Dayal, A.; Woolley, C.; Watson, B.; aj.: Adaptive frameless rendering. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, 2005.

[7] Halton, J. H.: Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, ročník 7, prosinec 1964: s. 701–702, ISSN 0001-0782.

[8] Havran, V.; Bittner, J.: On Improving KD-Trees for Ray Shooting. *Journal of WSCG*, ročník 10, č. 1, únor 2002: s. 209–216.

[9] Lewis, J. P.: Algorithms for solid noise synthesis. *SIGGRAPH Comput. Graph.*, ročník 23, č. 3, Červenec 1989: s. 263–270, ISSN 0097-8930.

[10] Lext, J.; Assarsson, U.; Müller, T.: BART: A Benchmark for Animated Ray Tracing [online]. `http://www.ce.chalmers.se/BART/`, 2003-06-02 [cit. 2012-01-05].

[11] Lindeberg, T.: *Scale-Space Theory in Computer Vision*. Norwell, MA, USA: Kluwer Academic Publishers, 1994, ISBN 0-7923-9418-6.

[12] Mitchell, D. P.: Generating antialiased images at low sampling densities. *SIGGRAPH Comput. Graph.*, ročník 21, srpen 1987: s. 65–72, ISSN 0097-8930.

[13] Niederreiter, H.: *Random number generation and quasi-Monte Carlo methods*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992, ISBN 0-89871-295-5.

[14] Schlömer, T.: *Frameless Rendering*. Diplomová práce, Universität Ulm, 2006.

[15] Shannon, C. E.: Communication in the Presence of Noise. In *Proceedings of the IRE*, ročník 37, 1949, ISSN 0096-8390, s. 10–21.

[16] Shirley, P.; Morley, R. K.: *Realistic Ray Tracing*. Natick, MA, USA: A. K. Peters, Ltd., druhé vydání, 2003, ISBN 1568811985.

[17] Volders, P.: *Frameless Rendering*. Diplomová práce, Katholieke Universiteit Leuven, 2006.

[18] Wald, I.: *Realtime Ray Tracing and Interactive Global Illumination*. Dizertační práce, Computer Graphics Group, Saarland University, 2004.

[19] Wald, I.; Havran, V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). Sep 2006: s. 61–69.

[20] Zagier, E. J. S.: Defining and Refining Frameless Rendering. Technická zpráva, Chapel Hill, NC, USA, 1996.

[21] Zagier, E. J. S.: A human's eye view: motion blur and frameless rendering. *Crossroads*, ročník 3, květen 1997: s. 8–12, ISSN 1528-4972.

[22] Zagier, E. J. S.: Perceptually-Driven Graphics. Technická zpráva, Chapel Hill, NC, USA, 1997.

[23] Zagier, E. J. S.: Frameless Antialiasing. Technická zpráva, Chapel Hill, NC, USA, 1998.

Příloha A

Obsah DVD

Na přiloženém DVD jsou připojeny všechny potřebné materiály, které vznikly spolu s touto prací:

- zdrojové kódy demonstrační aplikace v jazyce `C++`, včetně *makefilů* pro rychlé zkompilování aplikace do spustitelné podoby,
- zkompilované spustitelné soubory aplikace,
- programová dokumentace,
- obrázky pro všechny pořízené výstupy,
- demonstrační videa pořízená při testování jednotlivých částí metody,
- zdrojové kódy této zprávy v LaTeXu, elektronická verze této zprávy ve formátu `pdf`,
- testovací scény ve formátu `aff` a konfigurační soubory pro jednotlivé scény,
- soubor `README.html` s popisem instalace a použití demonstrační aplikace, nápovědou a popisem klávesových zkratek a popisem příkazů pro konfigurační soubor.